A Distributed Agent-Based Approach for Coupled Problems in Computational Engineering

Von der
Fakultät Architektur, Bauingenieurwesen und Umweltwissenschaften
der Technischen Universität Carolo-Wilhelmina
zu Braunschweig

zur Erlangung des Grades eines
Doktoringenieurs (Dr.-Ing.)
genehmigte

Dissertation

von
Jan Hegewald
geboren am 13. 5. 1977
aus Bremen

Eingereicht am	4. 4. 2011
Mündliche Prüfung am	28. 6. 2011
Berichterstatter	Prof. Dr.-Ing. habil. Manfred Krafczyk
	Prof. Hermann G. Matthies, Ph.D

2011

Abstract

Challenging questions in science and engineering often require to decouple a complex problem and to focus on isolated sub-problems first. The knowledge of those individual solutions can later be combined to obtain the result for the full question. A similar technique is applied in numerical modeling. Here, the software solver for subsets of the coupled problem might already exist and can directly be used.

This thesis describes a software environment capable of combining multiple software solvers, the result being a new, combined model.
The combined models considered here typically require the compute capabilities of multiple combined machines. Thus, the support for distributed hardware and parallel computing was one of the features which had to be considered early during within the implementation design. A basic communication framework has been chosen among several candidates to take care of the inter-process communication. This has then been used as a starting point to build the coupling environment. The integration of sub-models has been developed to allow the seamless use of already existing sub-model software (also known as legacy code). Two important design decisions were crucial at this stage: First, every sub-model keeps full control of its execution. This has been realised by wrapping each sub-model by a software agent. Second, the source code of the sub-model requires only minimal adaptation. This is possible, because the communication between all the sub-models has been implemented in a self-organising manner. Therein, the sub-models choose themselves when to issue communication calls, with no outer synchronisation mechanism required.
Each of the sub-models may have been tuned for optimal performance on specialised hardware. Thus, the coupling of heterogeneous hardware is supported as well as the use of homogeneous compute clusters. Furthermore, the coupling framework allows sub-solvers to be written in different programming languages. Also, each of the sub-models may operate on its own spatial and temporal scales.

The next challenge was to allow the potential coupling of thousands software agents, being able to utilise today's petascale hardware. For this purpose, a specific coupling framework was designed and implemented, combining the experiences from the previous work with additions required to cope with the targeted number of coupled sub-models.
The large number of interacting models required a much more dynamic approach, where the agents automatically detect their communication partners at runtime. This eliminates the need to explicitly specify the coupling graph a priori. Agents are allowed to enter (and leave) the simulation at any time, with the coupling graph changing accordingly.

Along with the regular testing, both frameworks have been used to realize several coupling scenarios. Among them the coupling of fluid and structure solvers, the biomedical appli-

cation of in-stent restenosis and a massively parallel fluid simulation. The first framework has been released to the public domain and is already being used by other European research groups.

Kurzfassung

Da viele Problemstellungen im Ingenieurwesen sehr komplex sind, ist es oft sinnvoll, sie in einzelne Teilprobleme aufzugliedern. Diese Teilbereiche können nun einzeln angegangen und dann zur Gesamtlösung kombiniert werden. Ein ähnlicher Ansatz wird bei der numerischen Modellierung verfolgt: Komplexe Software wird schrittweise erstellt, indem Software-Löser für einzelne Bereiche zuerst separat erarbeitet werden. Eventuell liegen diese schon als fertige Softwaremodule vor und können direkt Verwendung finden.

In dieser Arbeit wird eine Softwareumgebung beschrieben, die eine Vielzahl von unabhängigen Software-Lösern kombinieren kann (Kopplungs-Framework). So entsteht dann ein neues, gekoppeltes Gesamtmodell.
Hierbei benötigen die gekoppelten Modelle in der Regel die Kapazität von mehreren verbundenen Rechnern. Die besonderen Anforderungen hierfür (Kommunikation innerhalb verteilter Hardware) waren maßgeblich für den ersten Schritt: Es wurde eine bestehende Softwarebibliothek gesucht, die als Grundlage für die Kommunikation zwischen den einzelnen Software-Modellen dienen kann. Weiterhin wurde die Anbindung der Teilmodelle an die Softwareumgebung derart gestaltet, dass bestehende Software-Löser verwendet werden können, ohne diese umzustrukturieren. Dies wird durch zwei wesentliche Designkonzepte möglich: Jedes Teilmodell verhält sich weiterhin wie ein selbständiges Programm. Hierfür wird jedes Teilmodell in einen Software-Agenten gehüllt. Zur Kopplung sind lediglich minimale Ergänzungen am Quellcode des Teilmodells nötig. Möglich wird dies durch die Struktur der Kommunikation zwischen den Teilmodellen. Sie lässt den Modellen die Kontrolle über die Kommunikationsaufrufe und benötigt zur Synchronisation keine Einflussnahme einer übergeordneten Instanz.
Manche Teilmodelle sind für den Gebrauch mit einer speziellen Hardware optimiert. Daher musste das Zusammenspiel unterschiedlicher Hardware ebenso berücksichtigt werden wie homogene Rechencluster. Weiterhin ermöglicht das Kopplungs-Framework, dass unterschiedliche Programmiersprachen verbunden werden können. Wie schon der Programmablauf, so können auch die Modellparameter, etwa die räumliche und zeitliche Skala, von Teilmodell zu Teilmodell unterschiedlich bleiben.

Als nächsten Aufgabenbereich behandelt diese Arbeit eine Vorgehensweise um tausende von Software-Agenten zu einem Groß-Modell zu koppeln. Dies ist erforderlich, wenn die Ressourcen heutiger Petascale Rechencluster benutzt werden sollen. Hierzu wurde das bisherige Framework neu aufgelegt, da die große Anzahl von zu koppelnden Modellen einer wesentlich dynamischeren Kommunikationsstruktur bedarf: Bei der gewählten Umsetzung erkennen und verbinden sich die Kommunikationspartner selbständig, wodurch kein expliziter Kopplungsgraph zum Programmstart mehr nötig ist. Die Agenten der

Teilmodelle können einer laufenden Simulation hinzugefügt werden (oder diese verlassen) und die globalen Kopplungsbeziehungen passen sich dementsprechend an.

Zusätzlich zu Tests während der Implementierung wurden beide Frameworks für unterschiedliche gekoppelte Simulationen verwendet. Darunter waren gekoppelte Fluid- und Struktur-Löser, die biomedizinische Simulation von in-stent restenosis und eine massiv parallele Strömungssimulation. Das erste Framework ist als freie Software veröffentlicht worden und wird bereits von anderen europäischen Forschungsgruppen genutzt.

Acknowledgments

My special thanks go to Manfred Krafczyk for being a very helpful supervisor, allowing me to focus the work mainly on my research topic to get deeply involved with it, being able to freely choose the methods, even if I referred to experimental tools from the cutting-edge of computer programming.

I would also like to express my thanks to Prof. Hermann G. Matthies, who kindly agreed to be a reviewer of this thesis.

I am grateful to my colleagues at the Institute for Computational Modeling in Civil Engineering (iRMB) at the Technische Universität Braunschweig for the expert knowledge which they always willingly shared with me. This helped me getting back on track whenever curiosity took me away from my topic.

I wish to thank all the former COAST people for bringing together different scientific fields of research and being helpful colleagues along the way.

This work has been done within the scope of the EU funded Information Society Technologies (IST) project *Complex Automata Simulation Technique*[1] (COAST).

[1] EU-FP6-IST-FET contract no. 033664

Abbreviations

The following list contains the long versions for abbreviations used repeatedly within this thesis.

ACL Agent Communication Language

API Application Programming Interface

COAST Complex Automata Simulation Technique

CPU Central Processing Unit

CxA Complex Automaton / Complex Automata

FIFO First In First Out

FIPA The Foundation for Intelligent Physical Agents

HLA High Level Architecture

HPC High Performance Computing

IP Internet Protocol

JADE Java Agent Development Framework

JNI Java Native Interface

JVM Java Virtual Machine

MABS Multi Agent Based System

MIMD Multiple Instruction, Multiple Data

MPI Message Passing Interface

MUSCLE Multiscale Coupling Library and Environment

OOP Object-Oriented Programming

P2P Peer to Peer

SIMD Single Instruction, Multiple Data

SI Système International d'Unités

TCP Transmission Control Protocol

Contents

1 Motivation

Numerical computing is an ever evolving field of scientific research. The growing capabilities of computer hardware are but one aspect which lets scientists try to solve ever more and more demanding models via numerical approaches.

Another driving force in computational engineering is the ability to build new software upon existing mature solutions: Previously explored numerical approaches are not only known as best practice, but the algorithms are also available in their software implementation. These are for example optimised data structures like hash and tree storage systems as well as libraries for geometry discretisation and partitioning algorithms or frameworks for graphical data visualisation. Using those existing and proven detail solutions can make the task of implementing a new numerical model more assessable, or even feasible at all.

Fully fledged numerical software is often tuned to operate best within a well chosen combination of programming language, compute cluster environment, compiler optimisations and a balance between memory consumption and CPU usage. A natural approach to build complex solver systems would be the combination of preexisting numerical solvers, letting each of those individual sub-solvers operate on its favoured hard- and software environment, with the result being a new, coupled simulation. To do so, the solvers must be able to execute independently, but still communicate with each other during the calculation. A general solution requires a dedicated coupling system, which has to provide seamless interfaces between the involved programming languages and distributed hardware components.

An important aspect of coupled simulations is the combination of submodels which operate on different spatial and temporal scales. To keep the models independent, the data has to be mapped between them. This offers the opportunity to choose the discretisation of each model separately, as it is appropriate for its current local problem size and its specific numerical error. Additionally, the overall computational speed can benefit from the possibility to keep coarse scale models at their discretisation level.

Within artificial intelligence research and social behaviour analysis, autonomously operating software modules are being used to deal with complex dependency settings. These interacting units are called software agents. Using these agents, the interactions between the modules can be treated as separate settings, yet still allowing an individual execution of each module.

This ability to detach the coupling dependencies from the models is considered to provide solutions also in the field of numerical high performance computing, where the coupling interactions are traditionally static and determined a priori.

2 Preliminary Considerations

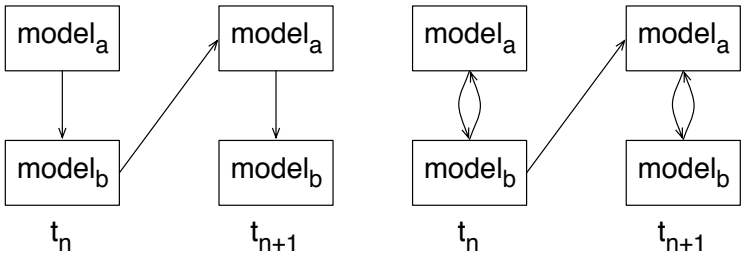
This chapter introduces more general aspects involved in the coupling of numerical models. Furthermore it outlines important aspects of distributed high performance computing (HPC), one of the major reasons why code coupling is necessary at all. Then strategies on how to couple, i. e. interconnect distributed software are discussed.

2.1 Model Coupling

Apart from the challenges of creating software to solve a numerical model, several aspects have to be considered regarding the numerical interaction and number of coupling dependencies.

Numerical Coupling Strategies

Different approaches exist to combine multiple numerical models into an assembled model. The basic interest is to perform a numerical integration for the whole model, i. e. solve the coupled differential equations [50]. The major approaches are the following:



(a) loose, explicit coupling (b) strong, implicit coupling

Figure 2.1: Partitioned transient coupling schemes

Loosely coupled partitioned systems, where the sub-models operate sequentially for each time step, waiting for results from another model as necessary [27]. This schema is also called explicit coupling (Figure 2.1(a)). It requires relatively small time steps to keep the coupling stable.

Strongly coupled partitioned systems, where the sub-models pass information in sequential order, as for the loosely coupled partitioned systems, but here the data

exchange will be iterated until a steady state in each time step has been reached [171]. The method is also referred to as implicit coupling (Figure 2.1(b)). Larger time steps are possible for this solution, but usually produce less accurate results.

Simultaneous monolithic systems, where the involved sub-models advance simultaneously in time and the joined model behaves as a monolithic model [89]. Here an identical discretisation is required for all involved sub-models.

Coupling Example

As an example for an explicit coupling, a heat transfer solver has been implemented. This example is part of the documentation for the developed MUSCLE framework (section 4).

To describe the steady-state heat transfer problem, the Laplace equation can be used:

$$\nabla^2 T(x,y) = 0 \tag{2.1}$$

Where T denotes the temperature. The solver uses an explicit finite difference scheme to compute the numerical integration.

The discretisation is based on an equidistant grid, where each vertex has a temperature value assigned. At the outer boundary, the vertices have a constant temperature throughout the simulation: A cosine profile at the top, a sine profile at the bottom and zero for the east and west boundaries (Figure 2.2).

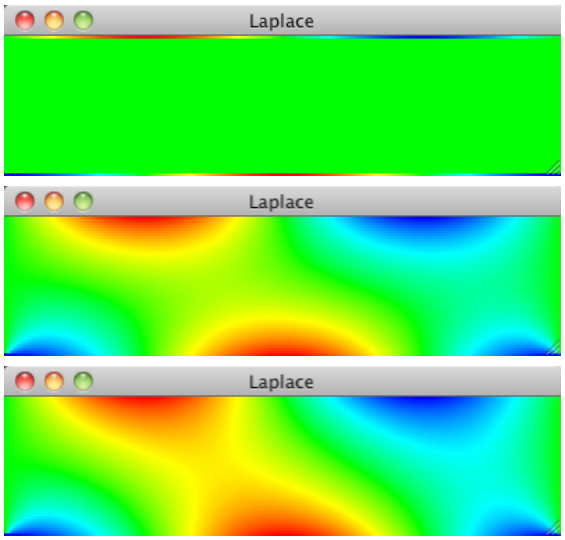
At $t = t_0$, the vertices in the bulk area are initialised with a temperature of zero. For each iteration of the computation, the new temperature values depend on the previous values of the vertex and its four adjacent vertices.

Figure 2.2 shows snapshots from the results as rendered by the GUI for a monolithic setup. The coloured output displays the temperature diffusion, which is continuously changing until it reaches a steady state. The colour indicates:

blue	−100 % temperature
green	0 % temperature
red	100 % temperature

In Figure 2.3, the same setup has been computed by two coupled partial simulations (west and east). Each of the two simulations calculates one half of the whole domain. Because each solver uses the same grid discretisation, the temporal and spatial scales are identical. Therefore no special kind of data mapping is required before message passing.

This setup is similar to a standard parallel computation implemented according to the programming paradigms of the message passing interface (MPI, page 44). Hence, a ghost-node column is used at the interface between the two grids to have all neighbour values locally available when calculating the new temperature values.



(a) begin

(b) middle (∼3000 iterations)

(c) steady (∼6000 iterations)

Figure 2.2: Output of sequential heat transfer simulation (the lower right corner shows resize marks from the GUI window)

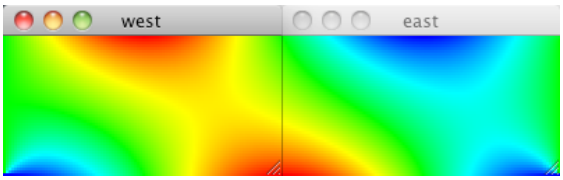
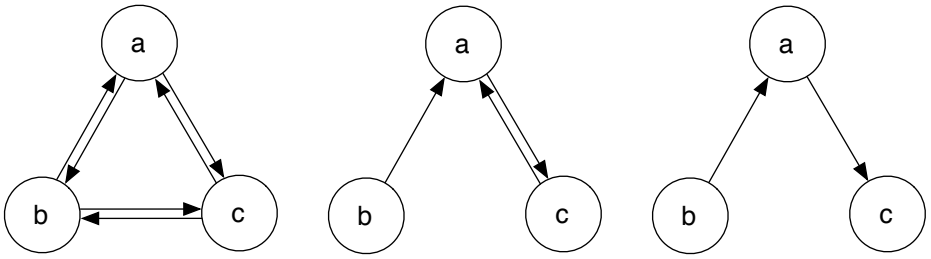


Figure 2.3: Output of parallel heat transfer simulation, ∼6000 iterations (the lower right corner shows resize marks from the GUI window)

Coupling Dependencies

In a general coupled setup, the sub-models of a simulation may depend on input from all or only some of the other models. Figure 2.4 shows several possible coupling dependencies for a scenario with three sub-models. The more other models a given model depends on, the longer it has possibly to wait for input during execution. Because of this, it is preferred to keep the number of coupling dependencies as low as possible, which also reduces network load.



(a) all to all, $D_{\max} = 6$ (b) some connected, $D = 3$ (c) minimum number of dependencies to join three models

Figure 2.4: Examples for coupling dependencies between three models: $D_{\min} = 2$

In the setup displayed in Figure 2.4(a), all sub-models $m = 3$ depend on all other sub-models $(m - 1 = 2)$, leading to the maximum number of $D_{\max} = 6$ dependencies. For the general case, this may be calculated as follows:

$$D_{\max} = m \cdot (m - 1) \tag{2.2}$$

There exist several possibilities to combine the minimum number of dependencies, with one combination shown in Figure 2.4(c). This minimum number of dependencies $D_{\min}$, where all the models are still connected together can be determined as

$$D_{\min} = m - 1 \tag{2.3}$$

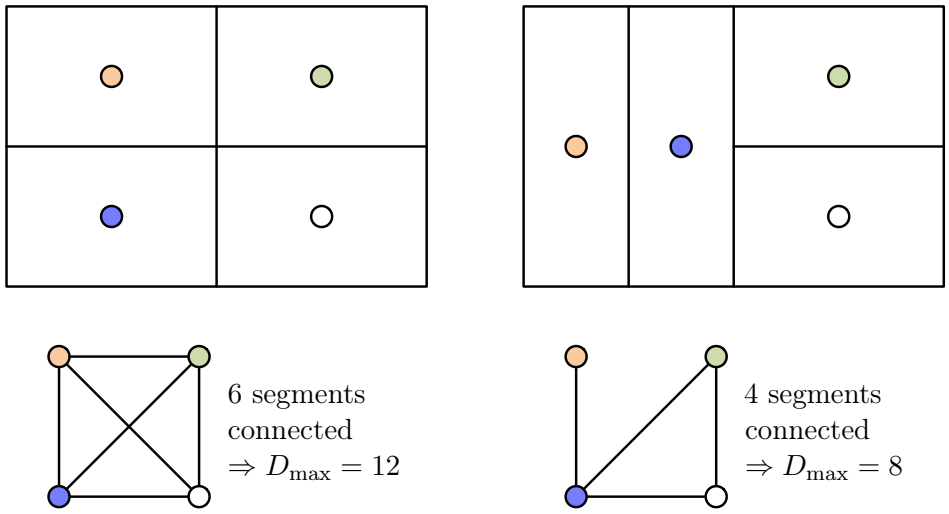
for an arbitrary number of sub-models m.

In between the range of $D_{\min}$ and $D_{\max}$ there also exist several possibilities how the models may depend on each other. For the example with three interacting sub-models, one case of combinations to have only three dependencies is depicted in Figure 2.4(b).

In some cases, the coupling dependencies between the sub-models are predetermined by the global numerical model. This is most likely the case for MIMD like solvers (section 2.3), where each sub-model serves a different purpose and therefore must be coupled according to the structure of the global model. This is e. g. the case for the coupled system of section 4.4.2, where three different solvers are involved: Their required dependencies

are fixed and the number of distributed parts is identical to the number of involved sub-models, which may not be chosen freely.

For classical SIMD solvers (section 2.3), the only difference of the individually connected sub-systems is how the whole problem domain has been partitioned. As these solvers often allow to split the whole domain into a *variable* number of partitions, the partitioning is usually automatically determined by a dedicated partitioning algorithm. Good partitioning libraries (section 2.3.1) try to keep the number of coupling dependencies low and should avoid the case of all to all dependencies. Figure 2.5 shows two naive approaches to partition a 2D domain into four segments. These $m = 4$ sub-models may have a maximum number of coupling dependencies of $D_{\max} = 12$ (equation 2.2, Figure 2.5(a)), whereas the minimum number is $D_{\min} = 3$ (equation 2.3, Figure 2.5(b)).



(a) maximum number of dependencies (b) reduced dependencies

Figure 2.5: Partitioning of a 2D domain into 4 segments

Multiscale Coupling

When each of the coupled sub-systems describes a different physical model, their spatial and temporal scales may be different too. Now the interaction of their spatial and temporal scales becomes another aspect of the modeling process.

In general it is preferable to let each sub-model continue to operate on its designated spatial and temporal scales: For some models it is not even an option so simply change their scales, as this would change the characteristics and quality of their calculation results. A common lower bound for the Δx_i of the spatial scales is the available memory. The computational load is typically a polynomial function of Δx. The lower bound for the Δt is also often constrained by hardware capabilities, as a very fine Δt would lead to an improper long calculation time. Also, very low Δt and Δx_i might introduce rounding

errors within the calculation. The reason is the limited (finite) precision of floating point values.

For a general coupling of arbitrary sub-models, three primary scenarios exist on how the scales of the individual sub-models are related:

- all the scales (i. e. spatial and temporal) of all sub-models are identical
- all scales of all sub-models are different
- a mixture of the above: some scales are different, some are identical

With two coupled models, each associated with a specific time scale and a one dimensional spatial scale, there are already four possible scale interactions, as shown in Table 2.1. Herein, m_1 and m_2 denote the first and the second model. Let further τ_m be the temporal scale of a model and ξ_m^d be the spatial scale for dimension d for a model.

$\tau_{m_1} = \tau_{m_2} \wedge \xi_{m_1}^1 = \xi_{m_2}^1$	all scales are identical
$\tau_{m_1} \neq \tau_{m_2} \wedge \xi_{m_1}^1 \neq \xi_{m_2}^1$	all scales are different
$\tau_{m_1} \neq \tau_{m_2} \wedge \xi_{m_1}^1 = \xi_{m_2}^1$	temporal scales differ, spatial scales are identical
$\tau_{m_1} = \tau_{m_2} \wedge \xi_{m_1}^1 \neq \xi_{m_2}^1$	temporal scales are identical, spatial scales differ

Table 2.1: Possible scale interactions for two models with one temporal and spatial scale

There have been a few approaches to find a formalism for the coupling of multiscale models [91, 79, 172, 81, 80].

A general definition is introduced by Hoekstra et al. [80], where each scale is defined by a range, i. e. from the current δt to the full duration covered by the simulation (T). The presented approach is called *Complex Automata theory* and provides a mechanism to describe multiscale model coupling problems. Therein the combined multiscale model is called Complex Automata (CxA) and the participating models are being mapped to a 2D graph according to their temporal and spatial scales. According to [80], this graph is called *scale separation map*.

Unlike the other solutions, which are targeting at very special applications [91, 172], the CxA theory describes a generic methodology to deal with a composed multiscale model [80, 79]. The term CxA describes a generic combination of cellular automata or multi agent based models, whose scales may be completely separated in space and time.

Regarding the spatial and temporal scales a coupled model operates on, these are inherited from the involved sub-models. For a minimum scenario with two coupled models M_0 and M_1, there may be two distinct spatial scale ranges ξ_0 and ξ_1 within the coupled setup. Also the temporal scales τ_0 and τ_1 of M_0 and M_1 may be different.

The minimum of the ξ range is denoted by δx, which is the finest spatial scale of the model. The maximum of ξ is limited by L, the spatial extend of the full domain:

$L = \delta x \cdot$ number of cells.

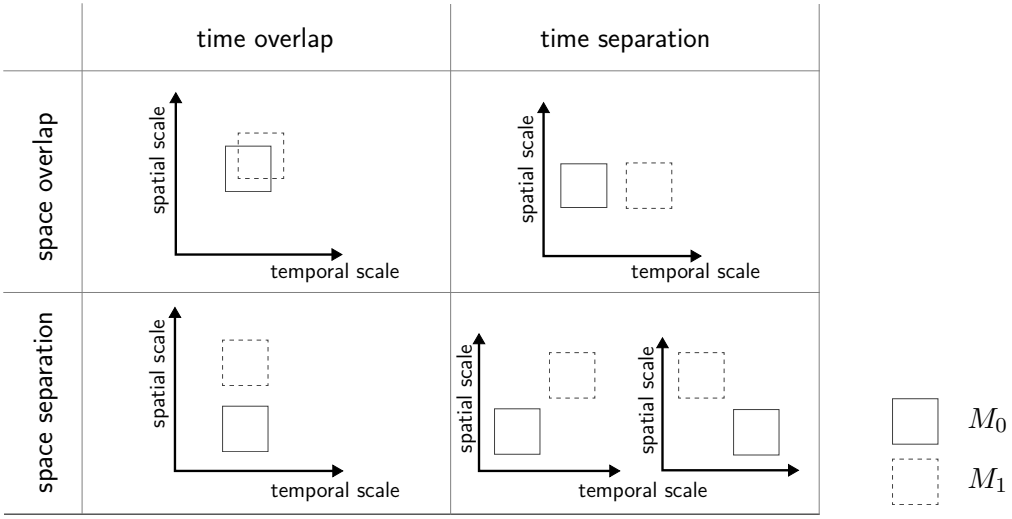
$$\text{boundary } \Gamma_\xi \text{ for a sub-model spatial scale} \qquad \delta x \leq \xi \leq L \tag{2.4}$$

For the temporal scale τ, the lower boundary is given by the δt (i. e. a time step). The total time T covered by a model is the upper boundary for the temporal scale.

$$\text{boundary } \Gamma_\tau \text{ for a sub-model temporal scale} \qquad \delta t \leq \tau \leq T \tag{2.5}$$

An advanced adaptive model may change the bounds for ξ and τ at runtime, which would lead to a δx, L, δt and T depending on the current execution time of the model.

For the both space and time scales of two coupled models, the range of ξ or τ may overlap or be completely separated (Figure 2.6).



modified from [80], Table 1

Figure 2.6: Possible scale interactions ($\sigma = 4$) when coupling two models (M_0 and M_1)

2.2 High Performance Computing

High performance computing (HPC) typically denotes numerical computations which require an extraordinary amount of CPU time and/or memory capacity. Those computations are then carried out on specialised hardware, e. g. compute clusters. Herein the CPUs and memory capabilities of a large-scale group of computers can be utilised by the numerical software. As the software has to integrate all the individual machines of such a compute cluster, this technique is also called distributed computing.

2.2.1 HPC Solvers in General

The vast majority of HPC solvers follow a common implementation strategy: they are implemented using either the Fortran, C or C++ programming language and internally utilise MPI to exchange messages between processes. Often, even the parallel implementations are being launched from a single monolithic binary and usually offer no means for further external interaction.

A widely applied usage of those solvers is to alter the configuration file, launch the executable and then post process the dumped results.

This common approach has the advantage, that external developers with no relation to the code development can still configure, compile and run the code by defining the target scenario in the configuration file. Configuring a whole calculation from a fixed configuration file has the advantage to fit into the concept of batch systems, where the individual application runs are submitted to a queue. They are executed as soon as the compute cluster has sufficient free resources. The big drawback with this classical configuration file approach is that the whole calculation scenario has to be anticipated in advance, as there is no further interaction possible during the calculation.

2.2.2 Parallel Computing

The purpose of parallel programmes is to split the programme execution into chunks which can be processed more or less independently. This way, they can be executed concurrently at the same time, each providing a part of the final overall result. There would thus be a theoretical speed gain by a factor equal to the number of programme chunks (section 2.2.4). A simple example would be the parallel computation of a brute force solution to calculate the sum of numbers from 1 to 100 by just adding up all the values.

The sequential (i. e. non parallel algorithm) would be the one from Listing 2.1.

```
1 int sum = 0;
2 for(int i=1; i<101;i++){
3     sum+=i;
4 }
```

Listing 2.1: Calculate the sum of numbers with a brute force algorithm written in C++

To be able to calculate this sum algorithm with two concurrently operating programme chunks, it could be split as shown in Listing 2.2.

chunk A

```
1 int sumA = 0;
2 for(int i=1; i<51;i++){
3     sumA+=i;
4 }
```

receive result from B

calculate total sum (e. g. within chunk A)

```
5 int sum = sumA+ sumB;
```

chunk B

```
1 int sumB = 0;
2 for(int i=51; i<101;i++){
3     sumB+=i;
4 }
```

pass result to A (involves network transfer for distributed memory systems, section 2.2.3)

Listing 2.2: Parallel version of sum algorithm from Listing 2.1

When computational calculations are being run, there are almost always constraints regarding the available hardware resources. Such resources consist of the following major components:

- CPU speed (may also be limited by the memory bandwidth or cache sizes)
- available memory (RAM)
- available disc space
- available bandwidth between these components

There are rare circumstances where a computation would not benefit from more/faster available resources, e. g. if decoding a video to be viewed by a human: if the decoding is fast enough to display the video at its designated speed, the hardware resources can be regarded as sufficient.

Modern CPUs can carry out multiple computations at the same time, without increasing the duration required to finish one of the individual tasks. Computations which are limited by CPU speed can use these parallel processing to their advantage and split their main task up into several sub-tasks which can be executed simultaneously. The general approach for such a software is to

1. split the problem into sub-problems suitable for individual tasks
2. set up dependencies between the sub-tasks
3. compute sub-tasks (which may depend on other sub-tasks)
4. create a joined result

The items 1 and 4 are also referred to a preprocess and postprocess steps. They are usually executed sequentially and thus do not benefit from parallel execution.

2.2.3 Distributed Computing

A vast improvement regarding the efficient use of hardware resources can be achieved with distributed computing. With distributed computing, the parallel tasks will be executed on different machines, requiring the data to be split to the different address spaces of RAM. Here, as determined by the hardware, the programme logic and data layout have also to be *distributed*.

Where shared memory parallel computing allows less complex (and less effective) parallelisation, distributed memory parallelisation requires a completely different programme logic and is challenging to implement, even for seemingly simple algorithms.

2.2.4 Performance Characteristics

The expected benefit of parallel programme is to reduce the time required for a computation. This duration should drop proportionally with the amount of available CPUs.

Distributed parallel programmes are spread to multiple independent machines, each having a separate memory (RAM). The combined RAM is now available for the programme, which makes it feasible to allocate larger problem sizes with an increasing number of machines.

Hereby, the programme aims to increase the maintained problem size and the computational speed linearly with the number of machines (i. e. "size" of a compute cluster). In practical use, this will be prevented by different effects:

- not all programme parts can be parallelised
- the maintenance of the parallel programme parts adds to the overall duration
- time for network traffic increases with more involved machines
- workload is not distributed evenly (load balancing)

The speed gain due to parallelisation can be expressed by the *speedup* of the parallel solver [132]. The speedup expresses a reference wall clock time T_r with respect to the wall clock time used for a number n of parallel jobs T_n, keeping the total problem size identical (i. e. *strong scaling*).

$$\text{speedup} := \frac{T_r}{T_n} \tag{2.6}$$

The reference wall clock time T_r is usually the time used for a sequential execution of the programme. For some computations it is not possible to determine the duration for a single CPU. This is e. g. the case if the problem size does not fit into the memory of a single machine. In these cases, the reference time T_r is determined for a (small) number of processes and the speedup is calculated relative to the reference number of machines.

Another measure for parallel performance gain is the *parallel efficiency*. To determine the parallel efficiency, the speedup is calculated relatively to the number of participating

processes:

$$\text{parallel efficiency} := \frac{T_r}{T_n \cdot n} \tag{2.7}$$

In some cases, the speedup of distributed programmes can be better than linear. This happens, if the distributed problem sizes become small enough to fit in a RAM region which has faster access due to ccNUMA (section 2.2.5) or if it even fits in the very fast caches between RAM and CPU (e. g. level 1 cache). This allows a much faster processing because the data is not being read from the slower main memory (RAM).

The so-called *scaleup* is a way to measure parallel speed without effects due to changing problem sizes. To measure the scaleup, the size of the calculated problem is increased accordingly with the number of processes. This strategy is also known as *weak scaling*. It creates a normalised problem size and tries to reduce cache effects.

$$\text{scaleup} := \frac{T_r}{T_n} \cdot n \tag{2.8}$$

With an increasing number of parallel jobs, the parallel efficiency of a programme drops. This is due to the fact that there is always some programme logic which can not be parallelised. The amount of this drop in parallel efficiency expresses how well a parallel programme *scales* with an increasing number of available machines, which can be described by the law of Amdahl [6]. If the ratio of sequential and parallel programme parts is called α, the number of concurrently executed chunks denoted as n, the law of Amdahl describes the maximum possible speedup (equation 2.6) according to the following equation:

$$\text{maximum speedup} := \lim_{n \to \infty} \frac{1}{\alpha + \frac{1-\alpha}{n}} = \frac{1}{\alpha} \tag{2.9}$$

Another influence can reduce the efficiency of a parallel programme: an uneven *load balance*.

Assuming that all machines in a cluster have identical hardware configurations, a good parallel efficiency can only be achieved if the workload of each process is identical. Otherwise all machines would have to wait for the machine with the higher workload to finish. As there is usually communication between the machines between each internal time step of the simulation, this delay occurs for every internal iteration.

An inversion of this relation is not necessarily true: A good load balance does not always lead to a good parallel efficiency. Redundancy of distributed data can lead to redundant processing, which puts all machines to 100 % load, but does not result in an optimal speedup for the overall computation.

2.2.5 Hardware for Distributed Computation

Today's large computer systems may be separated into two major categories:

- systems with collective memory access (shared memory systems)
- systems with multiple independent memory spaces (distributed memory)

In reality this distinction is not always that simple, as many hybrid approaches exits. The technique of shared memory systems is also called *uniform memory access* (UMA). Variants exist, where a CPU can access a dedicated (local) region of memory faster than the rest: *non uniform memory access* (NUMA) and a slightly different approach, *cache coherent NUMA* (ccNUMA). The UMA technique is sometimes not directly present in the hardware, but instead is abstracted by a software layer, which provides a global address space for parallel programmes.

Within a general shared memory system, all CPUs (or cores) share a globally available memory. This allows a relatively simple porting of sequential programmes to those architectures, as data layout and -storage do not need any modification. Special compilers for shared memory parallelisation (e. g. OpenMP [29]) can even perform this porting semi-automatically, only requiring minimal changes of the source code. This big advantage of shared memory systems comes with a drawback: The size of shared memory systems is more limited as compared to distributed memory architectures. For example the SGI Altix, a popular shared memory system, currently has its limits at 1024 CPU cores with 128 terabyte of memory (RAM) [152].

Distributed memory systems can be scaled to much larger amounts of CPUs and combined memory. This architecture is also called *no remote memory access* (NoRMA). They are often created with thousands of standard PC components, which come at a reasonable cost. These are connected with high speed network connections (e. g. InfiniBand or Myrinet [9]), to limit the communication overhead for parallel programmes. This way, the number of connected machines can grow to very large numbers. For example the Blue Gene/Q , planned to be finished in 2012, will have 98,304 connected machines with together 1.6 petabyte memory [13].

Programmes for distributed memory systems can not easily be created from existing sequential programmes. They require a completely different data layout, which has to be segmented for the now distributed memory regions. Explicit network communication is required to exchange data between the parallel tasks. The complexity of these programmes is much higher as compared to the shared memory model. A distributed memory programme may also be used on shared memory systems.

Multiple computers within a local area network may also be used as distributed memory systems, e. g. to use compute resources of company networks at night. These setups are also called network of workstations (NOW) [115].

Modern computers often use multiple CPUs, which themselves internally consist of multiple compute cores. Thus, systems which are purely distributed memory architectures are not really built anymore, as all the cores of a single machine share the same memory.

Another category of high performance computers are machines which use vector processors. These can *simultaneously* process elements from 1D ordered data sets (i. e. vectors). Those machines require a very special programme structure which typically operates on 1D data arrays. Among these special purpose machines which are in service today are for example the Cray Y-MP or NEC SX architectures. Because these machines are not very well suited for general purpose algorithms and are also expensive (in comparison to cluster systems) they are less widely used. When considering parallel HPC programmes, the above described scalar clusters (i. e. non-vector) are the most widely used parallel systems nowadays [127].

With the advent of new programming models for graphic processing units (GPU), high performance computing can also leverage this very specialised massively parallel hardware [101]. Using GPUs not for graphic output but rather for generic HPC programming is also called general purpose computation on GPU (GPGPU). The number of SIMD units on modern GPUs is rapidly increasing, with currently 3072 available in the AMD Cayman series. The drawback with GPGPU computing lies in its very specialised programming API. Like programming for vector machines, it only offers a limited functionality. Also, the available memory only is in the order of a gigabyte.

Currently, the fastest computer is the Chinese Tianhe-1A, a mixed architecture consisting of 14336 CPUs, each with 6 cores, and 7168 GPUs, each with 448 SIMD units. Together they offer a performance of 2.56 P flop/s.

2.3 Coupling Scenarios

Distributed programmes may be categorised into two major programming paradigms:

SIMD Single instruction multiple data (SIMD) programmes perform identical code logic on all machines simultaneously. Merely the processed data is different, e. g. they calculate results for different partitions of the whole domain. This programming model originates from the programming of vector processors, which in principle also simultaneously apply the same algorithms to different data items.

SIMD programmes are very typical and perfectly fit to homogeneous distributed hardware, i. e. cluster systems.

MIMD With multiple instruction multiple data (MIMD) programmes, the concurrently executed algorithms (i. e. CPU instructions) can be different. This is usually the case where the term *coupling* is being used, which implies to bring different software components together. Thus, executing different algorithms in parallel.

As these programming paradigms go hand in hand with the appropriate hardware capabilities, hardware may also be classified in SIMD and MIMD savvy systems. The original definition goes back to Flynn [54] and has been described from the point of view of the hardware capabilities. Herein, the two other combinations, SISD and MIMD are also mentioned, the latter being a mere theoretical construct. Single instruction single data (SISD) would be a sequentially (i. e. non-parallel) executed programme.

If looking at those programmes at a more fine grained level, some of the MIMD programmes do in fact operate similar to a SIMD programme for some stages of their lifetime. For example, a parallel programme might operate on a lattice, which is partitioned according to the number of available CPUs. This approach is common for cellular automata solvers. The same algorithm will be applied to almost all of the lattice cells (SIMD), but for some special cases, like boundary conditions, the algorithm might be different depending on the coordinates of the cell, making the programme execute according to the MIMD model.

Generally, implementing parallel programmes which require different instructions (MIMD) requires more effort than to develop pure SIMD algorithms and can not benefit from compiler support for semi-automatic shared memory parallelisation.

2.3.1 Static Coupling

Within traditional monolithic SIMD parallel programmes most parallel solvers use MPI [78] as the communication layer and thus are written in Fortran/C/C++, because the bindings for those languages are part of the MPI standard. The connections between the distributed tasks are explicitly wired within the code and there is no direct way to alter the connection dependency graph from the outside. For the majority of MPI solvers this is indeed not required: a usual scheme is to distribute according to a domain partitioning, where each parallel task will then communicate to the tasks responsible for adjacent patches of the domain.

This domain decomposition is usually done automatically before the actual parallel programme starts its execution, i. e. within the preprocess step. Simple partitioning algorithms split the domain in only one direction, treating every domain part equally (Figure 2.7).

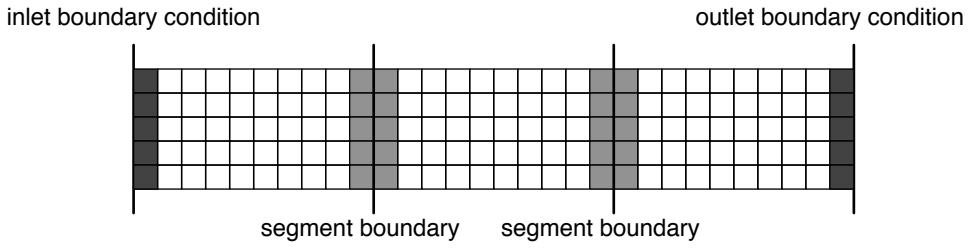


Figure 2.7: MIMD with bulk parts in SIMD

Another common way is to use a special partitioning algorithm, as e. g. the METIS

library [100]. METIS is popular for partitioning because of several aspects: it can be used as stand alone programme, or linked with other software. This is especially simple, as its source code is available for use and can thus be compiled for specific platforms. METIS is written in C, which can usually be linked to any kind of other language. Apart from these usability aspects, METIS offers several advantages regarding the result of the partitioning. It can be fine tuned to produce patches which produce a similar workload for the computation of the sub-domains. It further produces patches which have a small interface border, which is important to minimize the amount of data to be communicated between connected patches. METIS tries to keep the number of adjacent patches as low as possible to keep the number of coupling dependencies low (section 2.1 and equation 2.2).

Figure 2.8 shows the result of a METIS partitioning, where a uniform 2D lattice has been split into 8 patches. According to formula 2.2, this could lead to $8 \cdot (8 - 1) = 56$ dependencies, i. e. 28 pairs of adjacent patches. METIS was able to keep the number down to 12. For bidirectional interaction between all patches (equation 2.2) this would lead to 24 coupling dependencies among the patches (Figure 2.9).

Figure 2.8: Partitioned domain, 2D

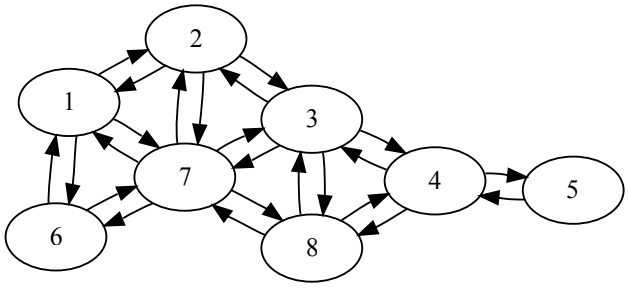


Figure 2.9: Connectivity graph of domain from Figure 2.8, 24 coupling dependencies

Another pattern of traditional coupled algorithms is the intermix of two or more models within a single codebase. These solvers usually consist of only a few coupling connections, e. g. one bidirectional dependency for a reaction diffusion solver.

2.3.2 Dynamic Coupling

With a statically configured coupling scenario one can address a lot of combined solver programmes. A more powerful approach is to account for a dynamic coupling of individual solvers. This leads to two categories: programmes, where the coupling graph will be determined at runtime, but only once during startup to remain fixed for the whole computation. Those are the a priori dynamic coupled programmes. The other category would be completely free dynamic setups, where the coupling graph and in fact also the number of involved models/solvers can change at runtime.

Dynamic coupling is also important for simulations with automatic load balancing or redundancy, where processes have to be shifted and migrated to other machines. This is also true for fault tolerant simulation scenarios, which become more and more important for very large scale clusters.

2.3.3 Multiscale Coupling

Solvers for different numerical models often operate on a specific spatial and temporal scale (section 2.1). Usually this is determined once and kept fixed during the calculation, but some advanced solvers operate on different scales at the same time or switch scales adaptively at runtime. Regardless on how a solver chooses its temporal and spatial discretisation, it should be able to do so in either a coupled setup or when used independently. This will guarantee the best results (regarding error, speed and problem size). An independent scaling of the coupled sub-solvers also leads to an optimised overall execution time and memory requirements, which most often is the sole reason to create multiscale simulations.

2.3.4 Coupling Different Programming Languages

The raw communication between distributed components boils down to the problem of how to identify the remote peer and then how to transfer the data across the network. The matter becomes more complicated, if software written in different programming languages has to be coupled.

Modern programming languages usually do have a built-in mechanism to be able to carry out some sort of remote procedure calls [67]. This is a high level approach for parallel communication and also allows message passing of bulk data. For other languages (Fortran, C, C++) one has to utilise a separate library for remote operations. In the HPC world this is almost always one of the MPI libraries.

Only a few remote communication solutions allow to connect to remote peers written in a different programming language. Even if technically possible with some frameworks (as with MPI), it is often not a robust solution in day to day usage. The ICE framework (page 41) and the CORBA specification (page 38) allow to interact across programming language borders almost seamlessly. A research project from 1998 has created a specialised MPI savvy prototype to allow heterogeneous communications [126].

3 Base Communication Library

Sophisticated numerical models often require parallel and distributed computing to cope with the hardware resources required by the calculation (section 2.2). They are also based on programming paradigms where existing numerical models may be coupled together to build more complex numerical models (section 2.1).

As the planned coupling framework was to be released to the public domain and will actually be used by other developers, it should use a software design which allows to implement individual parts of the whole framework independently. The implementation of these components can be separately tested and a component may be replaced with an alternative version. This is possible without the need to alter the other components. It is generally a good idea to utilise existing implementations, if they are proven and reliable. This generic programming concept is called component based software engineering [37]. If selected with care, third party libraries can greatly improve development progress and robustness of a software.

This chapter describes the process of choosing a basis software library which was the starting point for the coupling framework implementation.
Eventually, a multi agent based system called JADE has been selected as the supporting base library for the coupling framework.

3.1 Requirements

The requirements for the intended coupling of distributed heterogeneous numerical models arise from three major aspects: First, there are the many hurdles involved with distributed computing (section 2.2), which have to be addressed. Secondly, the interaction of different numerical models must consider coupling strategies (section 2.1) as well as the interaction between different spatial and temporal scales (section 2.3.3). The third challenge is the integration of various kinds of solvers, probably written in different programming languages. Some solvers may even require a special kind of hardware, like a large compute cluster or a vector machine (section 4.4.2), for optimum execution.

A straight forward integration of existing legacy codes is also mandatory, because a complete rewriting of software solvers should be avoided to build the combined model: Being able to use proven existing code should always be preferred to write a new one [74, 155, 28]. This does not only preserve all the detail solutions and specific workarounds for problems already experienced in the past, but also enables to build coupled solvers for

new combined numerical models in reasonable time.

In order to be usable within different distributed hardware environments, the final framework should also allow a portable configuration of the global setup: which solver is to be executed on which machine, maintenance of the configuration files or interaction with batch queues of compute clusters.

Together, the mentioned topics lead to the following list of required functionalities:

- parallel execution of the sub-models
- span multiple machines (distributed computing)
- allow heterogeneous hardware and operating systems
- allow different programming languages
- be able to couple 64 bit and 32 bit architectures
- keep the spatial and temporal scales specific for each model
- simple integration of existing code
- scalability with the number of models and machines

All these issues have to be addressed by the coupling framework. From a technical point of view, software could be considered to be coupled as soon as two code bases are able to exchange data. For simulations targeting at homogeneous HPC systems, an implementation of the broadly supported MPI standard (page 44) could be used to perform the raw message passing.

If the calculation is spread across machines which are connected via the Internet, the data may better be exchanged using CORBA, ICE (page 38, 41) or using systems dedicated for grid computing. These solutions are also suited to deal with high latency network connections in strongly heterogeneous setups.

Here, HPC applications which may utilise a multitude of compute nodes should be considered the standard use case. Those nodes may form a hybrid assembly regarding their hardware and operating system, but are interconnected via a fast network connection.

The idea of the project COAST was to approach the problem from a point of view less common to HPC: A multi agent based system should be used to accomplish the coupling, evaluating the capabilities of software agents in complex multi physics HPC coupling scenarios (A.1).

A basis software library should be utilised to limit the required effort to build the envisioned agent based coupling framework. To accomplish the development of a generic coupling framework, this basis library should allow to build the coupling framework according to the paradigms of multi agent based systems (MABS, section 3.2). Additionally, the basis library should allow to release the coupling framework with an open source compliant license.

3.2 Multi Agent Based Systems

Multi Agent Based Systems (MABS) are specialised software frameworks to build systems of interacting, but largely independent software units, so-called *agents*. These systems serve a variety of purposes, reaching from simulating swarms and crowd behaviour, artificial intelligence research and highly modularised, loosely coupled distributed systems [173, 174].

3.2.1 The Agent Abstraction

Programming according to the multi agent approach, also called agent oriented programming, is a software paradigm that applies concepts from the theories of artificial intelligence to existing techniques in the field of parallel and distributed systems. When implemented, this leads to a collection of components called *agents* which are used to participate in the resulting multi agent based simulation (MABS) [11, 174].

For the general case, each agent is commonly characterised by autonomy, proactivity and the ability to communicate. Being autonomous, they remain mostly independent for the whole computation. In terms of implementation, this usually requires a specific thread of execution. If the agents are proactive, they can choose to perform a given task at any time during the simulation. Their communicative abilities allows them to interact with other agents to exchange information regarding the combined task.

By design, any agent should be able to directly communicate with any other agent (also bidirectional), at a chosen time. Therefore the favoured architectural model of a MABS is intrinsically peer to peer.

Using software agents as a programming paradigm has already been discussed in 1994 by [66, 64]. As the concept of agent based programming became more and more important in the scientific community, an international workshop “agent oriented software engineering” with annual conferences has been established in 2000 [33].

With agent oriented programming, one can naturally develop distributed systems. It may be seen as a special application of object-oriented programming (OOP). As with an object in OOP, each agent encapsulates a state, and other components of the system can communicate with it through a public programming interface (API). This communication may change the internal state of the agent. But in contrast to OOP objects, the public API of agents offers a very narrow set of methods to call on an agent instance. Also, there is no direct way to modify the agent state from the outside. Only the agent itself may modify its state, but it can do so on behalf of requests or notifications from outside this agent. This is usually achieved by putting such a request (or notification) message to a message queue of the agent, which can be processed by the agent in any order. Often, putting messages to the agents queue is the only public accessible method of the agent API. The following listing shows such a method for an agent of the JADE framework (page 3.3).

```
1  public final void postMessage(final ACLMessage msg) {
2     synchronized(msgQueue) {
3        if (msg != null) {
4           //#MIDP_EXCLUDE_BEGIN
5           myToolkit.handlePosted(myAID, msg);
6           //#MIDP_EXCLUDE_END
7           msgQueue.addLast(msg);
8           doWake();
9        }
10    }
11 }                                                    Copyright 2000 CSELT S.p.A.
```

Listing 3.1: Public API method which allows to put arbitrary messages to a JADE agent

Being an entity within a distributed system, remote agents may not invoke methods directly on the target agent object, because they do not have a hold on an instance of the receiver (i. e. no instance of the target agent, because it exists on a remote host). The sender of such a message rather relies on the message delivery mechanism of the agent framework to pass the message to the host of the target agent and call its `postMessage` (or equivalent) method locally.

On a very abstract level, this procedure is similar to the concepts of OOP. In a general sense, calling a method on an object is the same as sending a well defined message to it. The details about how this is carried out is specific to each programming language. A general concept is the following syntax, which is available in C++, Java, Ruby and many other OOP savvy languages:

```
1 receiver.message(arg0,arg1,...)
```

Listing 3.2: Sending a message to an object in the general sense of OOP

This translates to the following: send a message consisting of `message(arg0,arg1,...)` to the object `receiver`. With some less flexible languages as C++, it is determined at compile time if a message with the stated signature is legal to be send to the receiver. Other languages (e. g. Ruby) allow to pass messages to any target, like it is possible with agent oriented programming: The agents may receive any kind of message to their queue and are in complete control as how to process the message.

Technically speaking, the two approaches have the same effect: sending a message to a receiver. In the general OOP sense, the semantics of this are specified by the programming language. In the case of agent oriented programming, the details are specified by the agent middleware (which may be implemented in different programming languages).

With agent oriented programming, the receiving agent decides internally how to react to an incoming message. With the OOP approach, the API usually reveals how the receiver will probably deal with an incoming message. Also the caller can usually assume that the receiver will take action immediately. This is also different with agent oriented programming: the receiving agent may choose when to look and possibly react to the

incoming message. This makes the agent approach a loosely coupled, asynchronous system by design. The components of an OOP design are much more tightly glued together and have a strong dependency between each other. Agents may exist and have a valid state without any other agents. This does not generally apply to any arbitrary object of OOP.

3.2.2 Advantages of Multi Agent Based Simulations

Agent oriented programming is a technique which is well suited to build large scale distributed systems, as their dependencies are reduced as much as possible. Because the interfaces of the agents have no direct relation on each other, it is technically more feasible to develop dynamic interactions, even using different kinds of (i. e. heterogeneous) agents [51].

Complex software systems consist of several more or less tightly coupled subsystems that are organised in a hierarchical fashion i. e. the subsystems can contain other subsystems. It is argued that agent based programming is well suited to reduce the complexity of building such software systems [96]. The topic is influenced by existing research in areas such as parallel and distributed discrete event simulation [59] and object oriented simulation [141].

For discrete event simulation systems it is assumed that state changes in the simulation occur at distinct points in time and are caused by so-called events. On the other hand, there is the continuous simulation, where state changes occur continuously over time. Discrete event simulation can be subdivided into event driven and time driven approaches [122].

Event driven approaches manage a chronological list of events, which trigger activity in the simulation. The result now could introduce additional events.

Time driven approaches advance the world clock of the simulation in discrete, usually constant, steps. Every simulation sub-task has to check, if an activity is due for this point in time.

The nature of a MABS implicitly suggests to use the event driven approach, because synchronisation with a global time will always lead to performance losses. Forcing every component of a simulation to be synchronous in time with the rest of the simulation will also prevent scalability of the whole system. Good scalability is only possible, if the coupling library allows for peer to peer communications among the different tasks.

3.3 Software Library Survey

This section introduces several existing communication libraries, which have been considered as basis for the coupling framework (Table 3.1). Each of them is capable of handling data exchange within distributed systems. For some of the libraries alternative implementations exist, as they are implemented according to a general specification.

Some of them are broadly used, like MPI, CORBA or HLA, others are less common but

ASH	FIPA-OS	MPI
Cactus	FLAME	MSI
CCA	HLA*	PALM
CCAFFEINE	ICE	RCF
CCAIN	JADE	SCIRUN2
CoCos	Jadex	TACO
CORBA*	jaRTI	X-agents*
COUGAAR	Jini	XCAT
CTL	LEAP	
DIET Agents	Middlesim	

*general specification, multiple implementations exist

Table 3.1: Libraries and specifications for distributed systems which have been considered to provide the basis for the coupling framework

nevertheless reliable. The libraries are for example being used to build distributed high performance solvers running on homogeneous compute clusters (MPI, Cactus, FLAME). Others are targeted towards systems which run in unreliable networks and across a variety of hardware (Jini, JADE).

The libraries range from low level communication solutions (also called middleware, Table 3.2) to libraries with a high level of abstraction. The advantage of the low level libraries is that they are usually highly optimised for a specific task. The higher level libraries allow a more flexible programming and often provide a rich set of features, yet the high level libraries often have to be used according to a predefined programming paradigm, which may interfere with the goals of the coupling framework.

CORBA*	Common Object Request Broker Architecture
CTL	Component Template Library
ICE	Internet Communications Engine
MPI*	Message Passing Interface
RCF	Remote Call Framework
TACO	Topologies and Collections

*general specification, multiple implementations exist

Table 3.2: Considered low level communication middleware and protocols

The following frameworks have been surveyed at the end of 2006. Some may have matured since then, others do not seem to be maintained anymore.

ASH

The middleware A Simple HLA (ASH) is an implementation of the HLA specification (page 40). It is an open source project written in C++ [166]. The implementation has its focus on Linux operating systems, but may be used in other environments with some effort.

The ASH project does not seem to be maintained anymore.

Cactus

The Cactus framework [21] is a flexible coupling system to aggregate different software kernels.

As HLA, its main target is not MABS, but to couple a few particular solvers. The source code is freely available under the GNU Lesser General Public License and thus offers some extendability. Cactus is written in the C programming language and uses MPI as the internal communication library. The plugin mechanism of Cactus is called "thorns", which are mainly used to provide algorithms which can be shared with the simulation. The simulations accomplished with Cactus are often based on discrete grids, which are especially suited to benefit from the "thorn" architecture.

Cactus offers a minimal steering support, which can be used to change simulation variables at runtime.

CCA

The Common Component Architecture (CCA) is a specification for distributed software middleware [7, 36], which defines a standard component architecture for high performance computing. The specification is supported from the group called Common Component Architecture Forum [160]. As an exception to other distributed middlewares, CCA supports the use of remote peers written in Fortran along with C++, Java and Python support. A usable implementation of the CCA is the CCAFFEINE framework (see below).

The goal of CCA is to be able to build large scale applications by combining multiple individual software components. These components are connected via a so-called "provides port" and "uses port", the former being a kind of outlet for a service, which is consumed by a "uses port" (inlet).

CCAFFEINE

The CCAFFEINE framework provides a reference implementation of the Common Component Architecture specification (see above) [4]. There seems to be no recent activity on the project [3], whose major development has been done by the High Performance Computing Research Division at Sandia National Labs in Livermore.

To build an run the CCAFFEINE, a Java runtime with Java compiler and a Ruby interpreter are required in addition to a C/C++ environment [26]. Internally, the CCAFFEINE uses MPI for message passing.

A graphical user interface (GUI) exists to setup the component connections for a CCAFFEINE simulation. A simulation may also be configured without the GUI from a command line interface, which may be used in batch mode or interactively.

CCAIN

The *CCA In*tegration framework (CCAIN) is another implementation of the CCA specification. The project has been realised by the Los Alamos National Laboratory and has later been made available to the open source community [25]. The CCAIN project has been removed from the CCA website, as there seems to be no active project maintenance anymore.

CoCos

Originally developed to communicate with networks of sensors, the Coordinated Communicating Sensors library (CoCos) is a communication middleware with a relatively low level API. It is being developed by the Brandenburgische Technische Universität Cottbus with funding from the Deutsche Forschungsgemeinschaft (DFG). It is also suited to operate in unreliable high latency networks (e. g. wireless networks) and thus allows a very loose coupling of the distributed components [105, 106, 99].

CORBA

The Common Object Request Broker Architecture (CORBA) is a specification for a middleware to allow communications in distributed applications. The specification is developed by the Object Management Group consortium [136].

CORBA explicitly provides mechanisms to deal with heterogeneous environments. As such, it can also connect peers written in different programming languages. To do so, every interface that should be usable remotely must be declared with an intermediate interface description language. These portable interface descriptions can then be converted to a matching implementation of a supported programming language. Supported languages include C, C++, Java, Ruby and Python [139].

A very popular implementation of the CORBA specification is named TAO, which is written in C++ [147]. Being an open source project, TAO may be use without licensing costs and is available at [137].

COUGAAR

A research project of the Defense Advanced Research Projects Agency (DARPA) [42] led to a multi agent based framework called Cognitive Agent Architecture (COUGAAR) [73, 39, 159]. It is developed by BBN Technologies [87]. COUGAAR is entirely written in Java and uses a component model similar to the Java Beans technology [95, 168]. The software sources are available under an open source license (Cougaar Open Source License) from [40].

On specific ability of COUGAAR is its fault tolerant behaviour in situations where hardware resources may change unexpectedly.

CTL

Following similar mechanisms as CORBA, the Component Template Library (CTL) is a middleware to allow distributed communication, where remote calls may be invoked on so-called *components* [55]. When calling to the remote components, the CTL aims to provide a highly efficient mechanism to keep the overhead of invoking remote procedures as low as possible.

The library has been utilised to couple scientific numerical solvers, which have been run in HPC environments [118, 121].

The CTL implementation is written in C++ and is being used like a language extension, e. g. the standard template library (STL). It relies on generic template programming and thus generates code at compile time via the preprocessor. The library is available in the public domain under the LGPL.

In addition to the C++ components, the connection to modules written in C or Fortran is natively supported. A special version of the library exists to allow the use of Java based components [17]. The internal message delivery of the CTL may be exchanged to use e. g. TCP or MPI.

DIET Agents

The *Decentralised Information Ecosystem Technologies Agents* platform (DIET) strives to allow loosely coupled systems of software agents, where each agent is a very lightweight component [119]. Within the framework, these are being called “infohabitants” which reflects the original purpose of the framework: information management. The system can run multiple agents on a single or several machines. As the framework targets lightweight agents, it has been designed to allow to run $\mathcal{O}(10^5)$ “infohabitants” on a single machine.

The library has been implemented using the Java programming language during an EU funded project. There has been no public project activity recently.

FIPA-OS

The FIPA-OS is an agent based system which conforms to the agent interaction standard defined by the *Foundation for Intelligent Physical Agents* (FIPA) (section A.6). This agent framework is implemented in the Java programming language and provides the benefit of being FIPA compliant. Maintenance of this project is very limited and is does have a public community support [52]. The most recent available version is from March 2003.

FLAME

The Flexible Large-scale Agent Modelling Environment (FLAME) library has been developed to simulate biomedical cell simulations, where each cell is represented by a minimal software agent. FLAME also incorporates the x-agents specification (page 45).

At runtime, this agent steps through different stages of its internal life cycle. The interactions with other agents are specified with an XML like markup language, from which a code generator generates the corresponding implementation in the C programming language. Usually, the agents within a simulation follow very similar rule sets. The library uses MPI for internal message passing.

The library has been developed by Coakley during his PhD thesis [35, 34] and is further maintained by the University of Sheffield [53].

HLA

The High Level Architecture defines a standard to combine multiple independent simulators, also allowing interactive man in the loop simulations [41]. It has been created by the US Defense Modeling and Simulation Office. Individual simulators in HLA are being called *federates*, with a coupled simulation being a *federation*. At runtime, each single simulator is controlled by a "FedExec" module, which are themselves managed the HLA runtime infrastructure (RTI).

In HLA, a single RTI has to manage the whole coupled simulation. This allows to control the flow of messages between the individual simulators and enables HLA to allow fine grained control over the synchronisation of different clock speeds of the individual simulators. On the other hand, this central management imposes a bottleneck for large scale simulations: Every message is timestamped and has to pass the centralised communication component of the RTI, wherein the RTI enforces chronological activity of message flow [43].

Writing a HLA savvy component requires major changes in the existing code, as the HLA requires detailed control over the runtime of a single component (simulator). With respect to implementation, a simulator has to implement the *federate ambassador* interface, which declares over 40 functions to implement [153]. Rycerz states: "Currently, setting up distributed applications based on HLA requires tedious setup and configuration" [146].

The High Level Architecture (HLA) exists in different commercial and free implementations [45]. Just recently there has been some effort to port an agent based artificial intelligence simulation to HLA [108]. This version supports build-in distributed simulation across multiple machines, but the scalability problems due to the RTI still remain.

The mature HLA implementations are closed source and therefore not easy to extend.

These HLA implementations have been considered during the evaluation: CERTI, MITRE, jaRTI, OHLA, PITCH-HLA, Middlesim. At that time, none of them seemed to provide a RTI which is itself parallelised to reduce this bottleneck.

Although the HLA is considered a mature standard and new implementations appear

regularly, the standard guidebook dates back to 1999 [107].

ICE

Another middleware whose concept is based on similar ideas as CORBA is the Internet Communications Engine (ICE) [77]. It is developed by the software company ZeroC [177] which want to position ICE as a successor of CORBA [75]. As with CORBA, various programming languages are supported for each remote peer, e. g. C++, Java, Ruby and Python. The major parts of ICE are written in C++.

JADE

The Java Agent Development Framework (JADE) library is a software framework to make the development of distributed multi agent based systems feasible [11].

JADE has been selected to be supporting library around which the coupling framework MUSCLE has been built (section 3.3.2.3 and 4).

The main components of the JADE runtime are the software agents, an agent observer called agent monitoring service (AMS), which takes the role of a white page service. Secondly, a so-called directory facilitator (DF), which registers services which can be announced by the agents (i. e. yellow pages). When exchanging messages, agents communicate directly to each other (P2P), providing good scaling properties [31]. Agents sharing the same address space are grouped in one or more *container* modules, which maintain dynamic lookup tables for other agents in the platform (Figure 3.1). The containers are

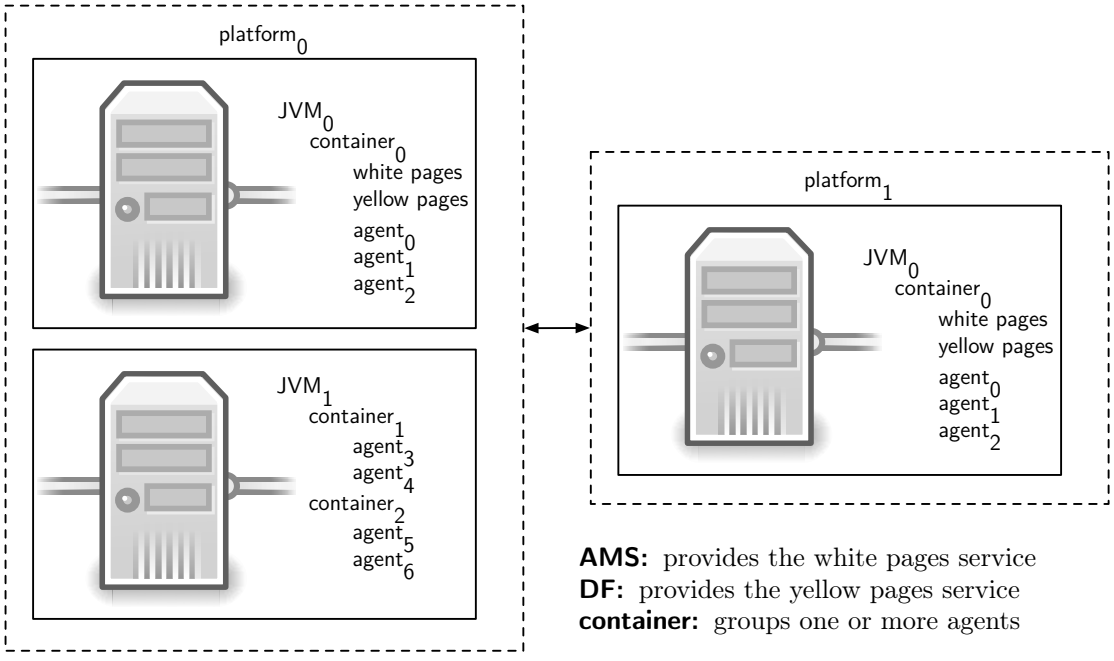


Figure 3.1: Elements of the JADE network mechanism

an efficient solution to let local agents share the lookup information. The lookup tables are required, because the agents may be added and removed to/from the platform at any time. Agents can also migrate to another machine at runtime, and messages sent to them will still reach their destination. Migrating agents can be important, if they have to access large amount of data: In this case, the data must not be copied over the network, which is usually a costly operation. It is even possible to migrate an agent to a machine where its codebase is not installed. JADE features a custom class loader, which can transfer the agent implementation to the target machine.

At runtime, agents may be suspended to a persistent state, i. e. the whole agent can be dumped to a file on the hard disc. Agents may be launched into a running platform using these files. JADE also offers a way to directly store agents to a database.

To add robustness to the platform, redundant versions of the DF and AMS can be created on several machines, which automatically synchronise their state.

If agents within the same container communicate to each other, they do not use the local loopback device as a dummy network, but instead JADE offers a more efficient shared memory access to the target agent.

The JADE framework offers full source code access due to its LGPL license model and may thus be freely extended. It also offers a plugin functionality to e. g. provide additional, shared functionality for the distributed agents or provide a different network protocol. Agents can be contacted from different network protocols at the same time. JADE and also a number of plugins are available from [92].

The whole framework is implemented in the Java language and offers full support for heterogeneous hardware and networks. It is fully conform to the FIPA standard, which defines communication mechanisms to interconnect different agent based simulations (section A.6). JADE has an active community and is widely used (section A.5). The JADE framework is actively developed and improved with frequent public releases (section A.4).

Jadex

The Jadex framework is a multi agent system, which has originally been developed to ease the development of complex large scale systems regarding artificial intelligence and behaviour research [16]. It is written in the Java programming language and focuses on the internal life cycle of the agents.

To carry out the agent administration and communication between individual agents at runtime, the JADE agent library is being used. With Jadex, the general purpose behaviours of JADE are greatly enhanced, allowing clustered settings of so-called beliefs, goals and plans.

Because it sits on top of JADE, Jadex also supports the FIPA standard. Jadex is actively developed by the Distributed and Information Systems Group of the Computer Science Department at the University of Hamburg [15]. Jadex is available in the public domain under the LGPL.

jaRTI

The jaRTI library is an implementation of the high level architecture runtime (HLA, page 40). The project has now been renamed and is called PoRTIco [117]. PoRTIco is supported by the Australian Defence Simulation Office and Calytrix Technologies. It is available under the Common Developer and Distribution License. The original jaRTI is available at [94].

Jini

The Jini is a network platform which allows to build large scale applications which are based on distributed services [133]. Services announce a remote handle (proxy) at a central repository, the lookup module of Jini. Components may connect to the distributed services via those proxies, which must first be obtained from the lookup repository through a so-called *discovery* module. So the original nature of Jini is that of a server-client architecture, without the need for a client to also act as a server.

To make the system more robust against failures, the central modules, lookup and discovery, may exist in redundant versions on multiple machines.

The whole framework is written in Java and uses Java RMI for network communication. It was originally developed by Sun Microsystems.

Jini is available in the public domain under the Apache 2.0 License [116].

Currently the project does not see much support and seems to be out of maintenance.

LEAP

The LEAP framework is a specialised version of the JADE agent framework. It is completely API compatible and any programme using JADE should be able to link to the LEAP library instead [11]. As the JADE library, LEAP is written in Java.

The LEAP version has originally been developed to better support mobile devices and brings features at the networking level which are significantly different from the plain JADE version. The network communication has been replaced with a custom layer which avoids the use of Java RMI. On top of this layer, a so-called split container mode may be enabled. This mode allows to fully integrate a software agent, even if the corresponding network connection can only be opened in one single direction (e. g. due to firewall restrictions). It also enables to reconnect to agent if the network connection was temporarily not available.

Additionally, the LEAP messaging layer may use specific port numbers on each machine, which is not possible with plain JADE.

Middlesim

The Middlesim framework is an implementation of the HLA specification (HLA, page 40). It is similar to jaRTI, but tries to provide a simplified interface. This way, it requires less effort and wrapper code to assemble HLA components to a combined simulation. The

Middlesim is written in the Java language and available under the Common Developer and Distribution License at [128].

The project seems to be out of maintenance.

MPI

The message passing interface (MPI) defines a software standard for low-level communication between multiple tasks, which are executed in parallel. It is widely use on SIMD like programmes which are designed to run on homogeneous PC clusters. Different open source implementations of the standard exist (OpenMPI [163], MPICH2 [129]) and hardware vendors often provide specific implementations for their cluster computers.

MPI can efficiently send data structures which are available in a continuous buffer (i. e. one single memory region), using one of the variants of its send routines:

```
1 MPI::Comm::Send(const void* buf, int count, const Datatype& datatype,
      int dest, int tag) const
```

Listing 3.3: MPI send, C++ binding

The receiving task has to explicitly connect to the send call, as it has to be done with socket programming:

```
1 MPI::Comm::Recv(void* buf, int count, const Datatype& datatype, int
      source, int tag) const
```

Listing 3.4: MPI receive, C++ binding

To be able to exchange messages with MPI, the tasks have to be grouped in a so-called "communicator" (MPI::Comm). Within the communicator, the tasks are being identified with an index value (**int** Comm::Get_rank()**const**).

MSI

Similar to Middlesim, the Multi-Simulation Interface (MSI) tries to simplify the process of creating HLA simulations [144]. It does so with implementing a model which is similar, but not identical to HLA. As HLA, it can deal with complex time synchronisation issues, i. e. synchronising the internal clocks of concurrently executing simulations and *human in the loop simulations*. The MSI can group simulations to allow hierarchical coupling scenarios. It is available to the public domain under the LGPL.

PALM

The Projet d'Assimilation par Logiciel Multiméthode framework (Palm) is a coupling library based on MPI. It is developed by the European Centre for Research and Advanced Training in Scientific Computation in Toulouse, France [18]. The tool can be used to build a simulation from several combined components (called "units"). These components are executed in sequence and the data between them is passed via MPI. Hence, the

components are limited to be written in either Fortran, C or C++.

The PALM features a mature graphical user interface which allows to define the connections between the components. A feature of PALM is the ability to launch components dynamically at runtime, not requiring to launch all components simultaneously. This way, components can be started conditionally at runtime.

PALM does not seem to have a public community, but commercial training sessions are available. The software may be obtained from [162].

RCF

The Remote Call Framework (RCF) is a communication framework completely written in C++. This solution can only connect C++ peers. Its benefit is the direct integration in C++ code, without the need to write an interface in an intermediate language [112, 113].

SCIRUN2

The SCIRUN2 is a component coupling framework which is compatible to the Common Component Architecture (CCA, page 37) [178]. It is developed by the University of Utah [150]. The SCIRUN2 is implemented using the C++ programming language and is available to the public domain under the terms of the MIT licence.

The SCIRUN2 brings a rich graphical user interface to define coupling relations of the components and also to visualise the computed results.

TACO

The *T*opologies *a*nd *Co*llections library (TACO) allows to build distributed programmes which simultaneously apply the same execution logic to on all remote data sets, i. e. it is targeted towards SIMD parallel applications [134].

TACO provides a global view of the combined RAM of all connected machines, allowing to develop SIMD algorithm quite naturally. From the user point of view this is mainly achieved via so-called "global pointers". The network communication is internally done via e. g. MPI or TCP, which is utilised to provide a custom remote method invocation layer. Taco is well suited to apply collective operations (e. g. gather, scatter, map) on data sets which live in distributed memory (i. e. different machines).

Taco is written in the C++ programming language and is being used as a template library, which integrates seamlessly with C++ programming, similar to the standard template library (STL).

Taco has been developed at the Brandenburg University of Technology in Cottbus [109].

X-agents

The X-agents specification is a formal concept to develop multi agent based systems [84]. It brings the theory of stream X-machines to agent based simulation [83]. Herein,

an approach similar to finite state machines is being used to combine different agent behaviours to process a combined simulation. The behaviours are being executed in a context which can store data. This custom data may also have an influence on the state changes.

XCAT

The XCAT is a library to connect distributed components according to the CCA specification (page 37). It focuses on bringing components from web services to grid computing: The XCAT runs on grid systems and uses XML messages which are passed via the SOAP communication protocol. SOAP is commonly used in web based services.

XCAT has been developed by the University of Indiana [61] and continues a predecessor implementation called CCA Toolkit (CCAT) [14].

The main XCAT version is written in Java and is also called XCAT-Java [164]. A C++ variant exists (XCAT-C++), but it is out of maintenance.

3.3.1 Library Comparison Results

In order to get the first impression about a particular software library, the quality of their documentation has been examined and checked against the list of required features (section 3.1). In a next step, the software has been compiled and installed and a simple *Hello World* example has been compiled and executed.

The following measure was to check the current development state of the software. For example, if there are known problems mentioned in the frequently asked question list. If yes, this could hint to a somewhat sloppy quality management, as the developers describe workarounds to problems which arise frequently, rather than attending to them within the code base. At the same time, the corresponding bug reports have been inspected to better estimate the severity of a problem.

Another important aspect is the available community support (mailing lists, forums, IRC channels). This is sometimes the only way to be able to cope with time consuming detail solutions (i. e. multi language support, compiler or linker problems). The niche products, like RCF, ASH or TACO have a very small community and the available support and solutions are limited. Most of the well known frameworks (e. g. MPI, CORBA, Cactus) have an active community and many common problems have already been encountered by the users and are thus discussed in the public. Surprisingly, the community support for HLA and its specific implementations is poor. Apart from a very basic documentation, the only helpful resource for HLA support seems to be the customer seminars offered be the vendors of commercial HLA implementations, e. g. [142].

Albeit being a commercial product, the free support for ICE is fairly good. The drawback only becomes obvious with more specific questions: Here the support team often stops the free support and refers to the commercial one instead, thus hiding the solution for

the public.

Among the list of competing software frameworks, the MPI based solutions (like Cactus) inherit the limitations for spanning heterogeneous networks.

In case of MPI, communication between different specialised MPI implementations still prove problematic.

Some libraries focus on connecting thousands of very low resource software entities and ensuring scalability (e. g. DIET Agents), others try to provide no limitation for the resources a single agent may use (e. g. JADE, HLA). The quality and stability of the different implementations range from stable (e. g. some HLA brands, Cactus, JADE) to incomplete or unstable (e. g. jaRTI).

From this preliminary survey, four libraries and specifications stand out and seem to promise some of the functionality required to build the agent based coupling framework for complex automata: Cactus, CCA, HLA and JADE.

Cactus for being a mature and robust implementation with good support and efficient peer to peer architecture (Table 3.3). Although Cactus uses the low level MPI standard as a transport layer, it offers an interesting feature set, like the steering support and its well designed plugin architecture. But in general, Cactus has its focus towards parallel programmes which are designed according to the SIMD paradigm.

CCA is a promising standard as it explicitly targets high performance computing with the intention to make it feasible to build large scale distributed programmes via component composition. Currently the CCA seems to lack a mature implementation and also the corresponding user base to offer support and provide feedback on the framework usability and robustness.

HLA has a lot of the required features already in place (Table 3.3). A unique feature of HLA is its built-in time management mechanism to synchronise all participating simulations.

It remains unclear if these features are only advertisements in the specification, or if they bring a benefit to the high performance computing programming practice. The HLA is not widely used in the HPC community and hence support is probably problematic when developing HLA programmes for HPC compute clusters.

JADE plays a special role as it is not explicitly designed with HPC computing in mind. Yet, the feature list is very complete (Table 3.3), the community is very active, the documentation is good and the source code seems to be extensible. JADE also offers a modular plugin architecture to extend the base functionality without the need to change the source code of the core implementation. It supports a great variety of hardware, even a version for PDAs and cell phones exist (JADE-LEAP-pjava and JADE-LEAP-midp).

✓ available
– not supported
? ambiguous

framework	extensible	heterogeneous hardware	multi network	multi language	scalable	slim architecture	steering support	agent support	standard plugin architecture	network security	mature	loose coupling	tight coupling	hierarchical coupling
ASH	✓	✓	✓	–	?	–	–	–	–	–	–	–	✓	–
Cactus	✓	✓	✓	–	✓	–	✓	–	–	–	✓	–	✓	–
CCA	✓	✓	–	–	✓	–	–	–	✓	–	–	–	✓	–
CCAFFEINE	✓	–	–	–	✓	–	?	–	–	–	–	–	✓	–
CCAIN	–	✓	?	–	?	✓	–	–	✓	–	–	–	✓	–
CoCos	✓	?	?	–	?	–	–	✓	–	?	–	✓	–	–
CORBA	–	✓	✓	✓	✓	–	–	–	✓	?	✓	–	✓	–
COUGAAR	✓	✓	✓	–	✓	–	–	✓	–	✓	–	–	✓	–
CTL	✓	✓	✓	–	✓	✓	–	–	–	?	–	–	✓	–
DIET	–	?	✓	–	?	–	–	–	✓	–	–	–	–	–
FIPA-OS	✓	✓	✓	–	?	✓	–	✓	–	?	–	✓	–	–
FLAME	–	✓	–	–	✓	✓	–	✓	–	–	–	–	✓	–
HLA	–	✓	✓	✓	✓	–	–	–	–	?	✓	✓	✓	–
ICE	✓	✓	✓	✓	✓	–	–	–	–	?	✓	–	✓	–
JADE	✓	✓	✓	–	✓	✓	–	✓	✓	✓	✓	✓	✓	–
Jadex	?	✓	✓	–	?	✓	–	✓	–	?	–	✓	–	–
jaRTI	–	✓	✓	✓	✓	–	–	–	–	–	–	–	✓	–
Jini	✓	✓	✓	✓	–	–	–	–	✓	✓	✓	✓	–	?
LEAP	–	✓	✓	–	?	✓	–	✓	✓	✓	–	✓	–	–
Middlesim	–	✓	✓	✓	✓	–	–	–	✓	?	–	–	✓	–
MPI	–	✓	–	–	✓	✓	–	–	–	–	✓	–	✓	–
MSI	–	✓	?	✓	?	–	–	–	✓	–	–	–	✓	✓
PALM	–	?	–	✓	?	–	–	–	✓	–	✓	–	✓	–
RCF	✓	✓	✓	✓	✓	✓	–	–	–	?	–	–	✓	–
SCIRUN-2	✓	✓	–	–	✓	–	–	–	?	–	–	–	✓	–
TACO	✓	✓	✓	–	✓	✓	–	–	–	?	–	–	✓	–
X-agents	–	✓	–	–	✓	✓	–	✓	–	–	–	–	✓	–
XCAT	–	✓	–	–	✓	–	–	–	✓	–	–	–	✓	–

Table 3.3: Feature of reviewed coupling software comparison according to requirements (section 3.1). A feature has only been marked as *available*, if its implementation is mature and it is explicitly supported.

3.3.2 Testbed for the JADE Library

The most promising framework from the above survey is the JADE framework. It seems to already support some of the required functionality and, first of all, it is a mature MABS library with an active community.

In order to test JADE to communicate data between numerical simulations, it has been used to perform a distributed numerical flow simulation. For this purpose, an already parallel Lattice Boltzmann flow solver has been utilised [58]. It operates on a discrete lattice and the domain will be partitioned according to the number of intended parallel jobs.

The code base of the legacy code is written in the C++ programming language. The original implementation is a complex system for fluid flow, multi phase and free surface simulations. It may work on uniform or non-uniform grids and the simulation domain can be fully distributed to multiple tasks to make use of distributed memory computation clusters [58].

This makes the testbed simulation a distributed SIMD like solver, as the calculation of each partition follows according to the same rule set. The final complex automata coupling framework has to be able to be more flexible: the default intended use is to couple several very different solvers ("hybrid peers").

3.3.2.1 Flow Solver Numerics

The field of computational fluid mechanics often uses the Navier-Stokes equations to describe the dynamics of fluids. Important variables herein are the pressure and velocity field of the fluid at hand.

Approximate solutions of these equations can be calculated using the Lattice Boltzmann method [123]. Its idea is to solve the fluid dynamics problem at a mesoscopic level. For low Knudsen numbers ($\frac{c\tau}{\mathrm{L}} \ll 1$), the achieved results are equivalent to the ones of the Navier-Stokes equations (see Chapman-Enskog analysis [30]).

The Lattice Boltzmann-method allows an efficient computation, even for complex geometries [63, 62]. Because the discretisation uses a local stencil, the method is particularly suitable for distributed parallel computation.

The flow solver which has been utilised for this coupling example uses a fluid dynamics solver whose implementation is based on the Lattice Boltzmann method.

In the used flow solver, this numerical method operates on a uniform grid. Each vertex within the grid holds a discrete number of distribution functions (f_i) for the microscopic particle distributions. Simulations in 3D often use 19 discrete directions $i = 1..19$, which is called D3Q19 model (19 directions in 3D). The corresponding directions $\vec{e}_i$ for those distributions are shown in Figure 3.2, where one of the particle distributions ($i = 1$) is quiescent in the middle of this grid vertex, i. e. $\vec{e}_1 = (0,0,0)$. For the D3Q19 model, the 19

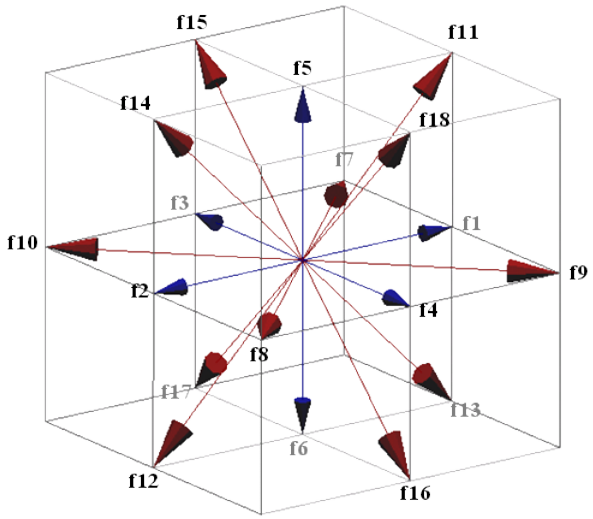


illustration by S. FREUDIGER

Figure 3.2: Discretisation stencil of the D3Q19 model

directions are given as follows:

$$\vec{e}_i = c \begin{pmatrix} 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \end{pmatrix} \tag{3.1}$$

In 2D, a common discretisation of the velocity directions is the D2Q9 model with $i = 1..9$.

Every one of the distribution functions f_i represents a microscopic fluid particle, which moves in the direction given by its vector $\vec{e}_i$. Each iteration step during the computation has to calculate the 19 distribution functions $f_i(\vec{x},t)$ at every lattice vertex $\vec{x}$.

From these results, the macroscopic values for density $\rho = \Delta\rho + \rho_0$ and velocity $\vec{v}$ within the fluid can be determined:

$$\Delta\rho = \sum_{i=1}^{19} f_i \tag{3.2}$$

$$\rho_0 \vec{v} = \sum_{i=1}^{19} f_i \vec{e}_i \tag{3.3}$$

With ρ_0 being the reference density. It is, without loss of generality, set to $\rho_0 = 1$ in the numerical model.

The computation of the Lattice Boltzmann method is composed of two major steps: The *collision*, where the interactions of particle distributions found at a grid vertex for a specific time are calculated. The other step is the *propagation*, where the newly available particle information is being transferred to the neighbour vertices.

During the collision step, equilibrium functions f_i^{eq} will be used to apply a relaxation to the distributions at each lattice point (equation 3.4). This requires to know the density and velocity (equations 3.2 and 3.3).

$$f_i^{eq}(\vec{x},t) = \omega_i \left[\Delta\rho + \rho_0 (3\frac{\vec{e}_i\vec{v}}{c^2} + \frac{9}{2}\frac{(\vec{e}_i\vec{v})^2}{c^4} - \frac{3}{2}\frac{\vec{v}^2}{c^2}) \right] \quad \text{(incompressible modell)} \tag{3.4}$$

$$\text{with} \qquad w_i = \begin{cases} \frac{1}{3} & \text{for } i = 0 \\ \frac{1}{18} & \text{for } i = 1..6 \\ \frac{1}{36} & \text{for } i = 7..18 \end{cases}$$

During the propagation step, the fluid particle distributions a being moved along their directions $\vec{e}_i$. As each time step Δt is normalised with $\Delta x = c\Delta t$, all particle distributions directly "arrive" at their adjacent grid points.

The procedure of collision and propagation can be described using the Lattice Boltzmann equation:

$$f_i(\vec{x} + \vec{e}_i\Delta t, t + \Delta t) = f_i(\vec{x},t) - \frac{\Delta t}{\tau}\left[f_i(\vec{x},t) - f_i^{eq}(\vec{x},t)\right] \tag{3.5}$$

Herein, the influence of the kinematic viscosity ν of the current fluid is given by the relaxation time τ.

The transfer between the Navier-Stokes equations and the Lattice Boltzmann equation can be made due to the following relations:

$$\text{speed of sound} \qquad c_s = \frac{c}{\sqrt{3}} \tag{3.6}$$

$$\text{kinematic viscosity} \qquad \nu = \frac{c^2}{6}(2\tau - \Delta t) \tag{3.7}$$

$$\text{pressure} \qquad p = c_s^2\rho = \frac{c^2}{3}\rho \qquad \text{(ideal gas law)} \tag{3.8}$$

The used flow solver is based on the multiple relaxation time approach (MRT), a variant of the Lattice Boltzmann method. Herein, the distributions f_i will not be modified by a single relaxation parameter. Instead a local transformation to an equivalent momentum space is used, after which a more costly collision operator is applied. As a result, the stability of the numerical model will increase. The extensions due to the MRT model are applied locally at each vertex and can thus be parallised effectively. The MRT approach is further described in [46].

3.3.2.2 Reconnecting The Original Solver

In the original code, the data between the adjacent segment boundaries will be exchanged using MPI communication. For the testbed application the distributed solvers with their corresponding domain segment have been wrapped in a software agent and then use the JADE message transport protocol (MTP) for network communication.

To integrate the sub-domains into the JADE agent environment, every simulation part (C++) is glued to a Java thread via the Java Native Interface (JNI, section 4.3.1). These threads execute the software agents in parallel under the umbrella of the JADE environment. Communication between the different parts of the simulation can now be done using JADE communication mechanisms. The MPI message layer of the original code was replaced to pass all messages via the corresponding agent belonging to each sub-domain. The JADE communication system now synchronises and transfers interface data of the corresponding subdomains.

In the simulation scenario, a Kármán vortex street has been calculated [98], once with the original solver and then using the JADE coupling. This way the results of the original code base could be used to validate the results of the JADE coupling. Figure 3.3 shows the flow velocity computed with 6 partitions, i. e. 6 parallel executed sub-solvers.

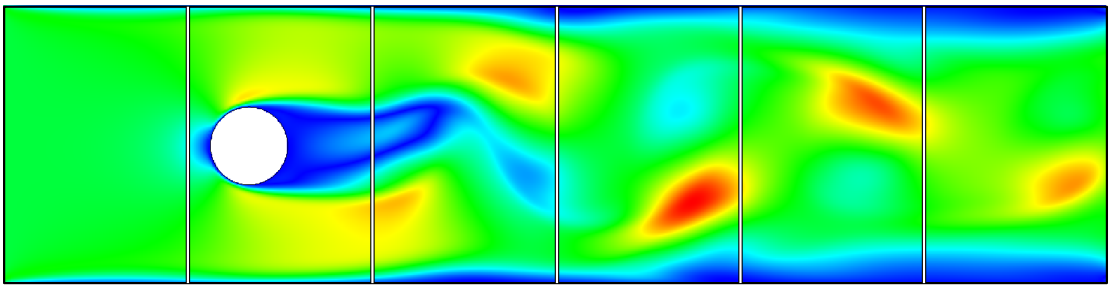


Figure 3.3: Distributed flow simulation utilizing JADE library, colours indicate the fluid velocity to the right, time $t = 5000\,\Delta t$

3.3.2.3 Testbed Conclusion

The JADE library has been used to successfully couple a parallel and distributed fluid flow simulation. It was also possible to use a C++ code with the JADE library, which itself is written in the Java programming language (section 3.4) and offers no direct language bindings for C++.

The JADE agent framework has been selected to provide the supporting middleware for the complex automata coupling framework. One of the main reasons was its flexibility regarding heterogeneous hardware, operating system and network support. JADE supports many features which are useful for the coupling framework (Table 3.3).

3.4 About Java

Java is a programming platform created by Sun Microsystems[1] and first released to the public in 1995. Programming with Java is well documented in several guidebooks [12, 168, 103, 167, 104, 48]. The name Java refers to two different things:

Language It refers to the specification of the programming language Java.

Runtime The name is also the name for the Java Runtime Environment (JRE) by Sun.

The Java language is comparable to C++, with a bigger emphasis on object oriented programming (OOP). Java has a garbage collector which is responsible for memory management and frees unused data automatically. Though having great resemblance to C++ on the source code level, there are some strict differences. Java is a save language: with the exception for calls to C/C++ libraries, it is immune to buffer overflow, stray pointers, variables with undefined state (i. e. if they are not correctly initialised). It is also not possible to overrun array boundaries. Other as with C++, the order of static variable initialisation is well defined in Java programmes and thus the behaviour is more robust.

Compiling Java source code and linking to third party libraries works without glitches in Java, whereas it can be a major inconvenience and time consuming in C++. At compile time, Java code is not directly transferred to machine code for a specific computer (e. g. executable or binary). Instead, the compiler generates an intermediate format, which can be used on any hardware, the *bytecode*.

The mediator between the bytecode and a specific hardware is the Java Virtual Machine (JVM). While executing a programme, the JVM uses a just in time compiler (JIT) to compile the bytecode to platform dependent machine code. As the JIT does exactly know the hardware it currently runs on, it can highly optimize the compilation for this very machine. While doing so, the JIT profiles the programme and knows if an optimisation pays off. This way the JIT can try aggressive optimisations without compromise, as it can always try again, now with additional knowledge. Of course the JIT compiler requires some CPU time itself, thus a very short lived programme may not benefit from this technique.

JVM implementations exist from different vendors (like open JDK or Sun JVM (Sun actually has multiple JVM implementations: HotSpot JVM [88] (written in C++ and C), the Maxine VM [161] (written in Java).

Java comes with a huge standard library which contains solutions for many common programming problems. About all the libraries which are part of the standard Java distribution are written in Java itself. The Java libraries together with the JVM are also called Java Runtime Environment (JRE) or Java Development Kit (JDK). The latter does also include the bytecode compiler and other development tools.

[1] Now owned by Oracle Corporation

Architectures: 64 and 32 bit

As with other programming languages, Java makes use of primitive datatypes to efficiently store characters, logical values as well as integer and floating point values. This is very similar to languages as C++, C or Fortran. A common problem with primitive integer and floating point values is their limited precision, which is a tradeoff with the required storage size. The usual choices are to select between 32 and 64 bit sized values. With languages as Fortran, C and to some extend also C++, the predefined size for a datatype may change when compiling on a different hardware architecture (64 versus 32 bit). With Java, this is always guaranteed to be the same specified precision and size (Table A.2), regardless of the underlying hardware architecture.

As coupling various software solvers may not only combine code from different programming languages, but also span different hardware architectures, this becomes an important matter.

Endianness

The byte (or bit) order within a data block (e. g. a 64 bit double) is called *endianness*. The two cases with which byte to begin are called big-endian and little-endian. With different hardware architectures come different preferences for the endianness. This only proves a problem when data is yielded from one mechanism and to be interpreted by the other one.

4 Multi-Scale Coupling for Complex Automata

The following chapter is dedicated to the developed coupling framework. It has been the software foundation within the EU project COAST, where it has been used to develop a coupled simulation according to the complex automata paradigm (section 2.1). Therein, multiple differing simulations should be coupled with an agent based approach (section 3.1).

Along with the development progress, the software framework has been released to the public domain under the name *Multiscale Coupling Library and Environment* (MUSCLE) [68, 70] with an accompanying guide for developers [69].

Implementation wise, the framework consists of three major parts:

Core library The core implementation of the coupling framework. It executes the sub-solvers and establishes connections between them. All the message passing is done via the core library. The MUSCLE core library uses the JADE library internally and has been implemented in the Java programming language. The description can be found in section 4.1.

Setup system The setup of a coupled MUSCLE simulation has to deal with different aspects: declare the sub-solvers to be used, define the interactions between all outlets and inlets, assign conduits for data mapping and also specify all the properties which specify the computation domain and details about the actual numerical problem. To be able to maintain all these settings in a feasible manner, MUSCLE uses a separate setup system, which is especially suited to express all configuration areas in a human readable and writable manner (section 4.2). This has been written in the Ruby programming language.

Native library The MUSCLE native library provides means to integrate sub-solvers written in C++, Fortran or C into a coupled simulation (section 4.3). This library has been implemented using the C++ programming language and establishes a bridge to the MUSCLE software agent.

4.1 MUSCLE Core Library

The core implementation of the MUSCLE library provides the functionality to develop and run distributed multiscale simulations consisting of different coupled solvers. Herein,

the JADE library has been used to provide the basic infrastructure for distributed software agents (section 3.3.1). From the JADE point of view, every sub-simulation of a coupled setup is a single software agent. The core implementation has been written using the Java programming language.

The implementation of the MUSCLE core has been written in a manner which makes best use of the used agent library JADE. Hence JADE is itself completely written in Java, the core parts of the coupling framework itself have been written in Java, too.

Apart from this, the main implementation language is always a tradeoff between several important topics:

runtime speed Which depends on CPU clock, cache sizes, memory bandwidth, compiler efficiency etc.

memory footprint Storage and management of large scale data structures, overhead for object allocation.

profiling support Analyze application at runtime to detect CPU and memory hogs or bandwidth critical parts.

feature set Richness of standard library and popular third party libraries.

documentation and support Usefulness of available documentation, code examples and community support. Also applies to any third party libraries being used.

compiler quality The compiler gives shape to the important features *runtime speed* and *memory footprint* and is thus crucial for the final application. It also determines how (or if at all) the framework can be used in future projects. Furthermore the compiler determines for/on which platforms/operating systems the framework can be build.

The MUSCLE library directly inherits three of its core components from the JADE library. The most prominent component of JADE is its basic software agent class, called `jade.core.Agent`. The JADE library itself uses specialised versions of this base agent e. g. for the implementation of the agent monitoring service (white pages) and the directory facilitator (yellow pages) (page 41). Three MUSCLE classes extend the base JADE agent to be able to facilitate the functionality provided by JADE and thus have a "is a" relationship to the `jade.core.Agent` in terms of software inheritance:

- One of these three MUSCLE components is the solver agent, which wraps and executes the main user code (i. e. software solver). Its relationship to JADE is depicted in Figure 4.1. This is *the* MUSCLE agent and it is further described in the following sections, especially section 4.1.4.
- As the white pages and yellow pages in JADE, MUSCLE uses a utility agent called *plumber*, which itself also is a jade agent (Figure 4.2). Its description can be found in section 4.1.3.
- The third MUSCLE component which inherits functionality from the base JADE

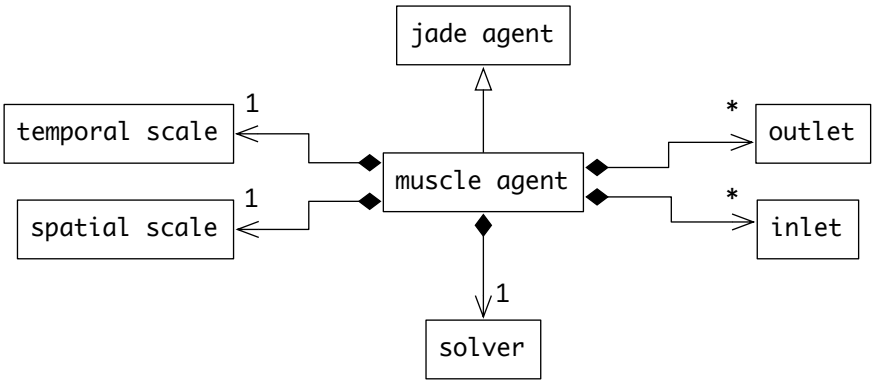


Figure 4.1: The MUSCLE agent inherits from the core JADE agent and thus allows a direct access to the JADE functionality

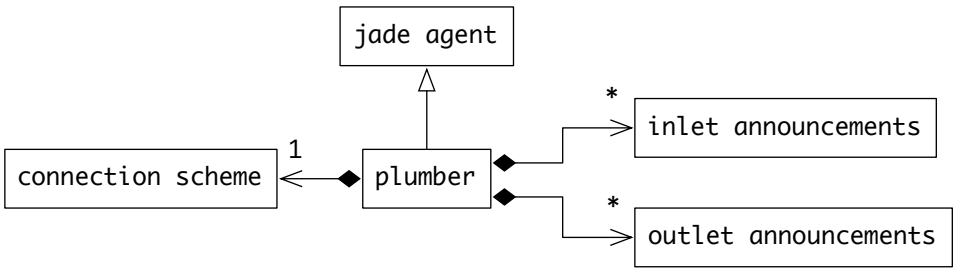


Figure 4.2: The plumber is a MUSCLE agent which has an "is a" relationship to the JADE agent

agent is the conduit implementation (Figure 4.3). Conduits are responsible to map the sender data to a format appropriate for the receiving solver (section 4.1.2).

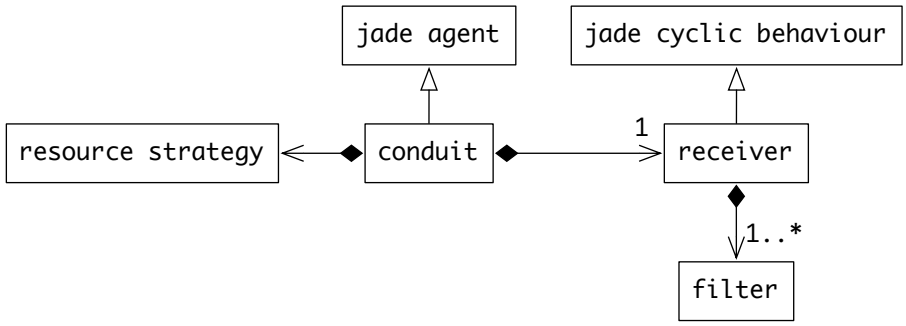


Figure 4.3: The conduit inherits functionality from a JADE agent and uses a recurring behaviour to pass messages to the receiving agent

4.1.1 How MUSCLE Works

The coupling framework makes it feasible to combine multiple existing simulations, letting them contribute to the solution of a bigger, combined problem. At runtime, the individual sub-solvers may communicate with each other to exchange intermediate results.

Every single sub-solver operates under the hood of a software agent and all the solver logic is encapsulated by the enclosing agent. This is also true for data belonging to a solver: it is wrapped and protected by the agent. The agent communicates with other agents on behalf of its underlying solver and is responsible to perform all the operations required to establish the necessary network connections. Looking from outside the agents, the network activity has no knowledge about the internals of a sub-solver and thus may be developed independently. From the communication point of view (i. e. from "outside" the agents), the sub-simulations are a network of interacting software agents (Figure 4.4). Inside the agents, the sub-solvers do the numerical calculation and communicate with other sub-solvers via the corresponding agents. Each agent has an arbitrary number of inlet/outlet slots to do so (Figure 4.5).

In some cases, these sub-solvers will be newly written to be part of a coupled simulation. As such it would be easy to do the implementation according to the terms of the coupling library:

- implement the given mechanism to automatically launch the sub-simulation
- pass the corresponding configuration file while launching
- use the frameworks' preferred data types within the sub-solver
- identify the other communication partners with the semantics of the coupling framework (e. g. IP address/port with TCP or communicator/rank combination with MPI)

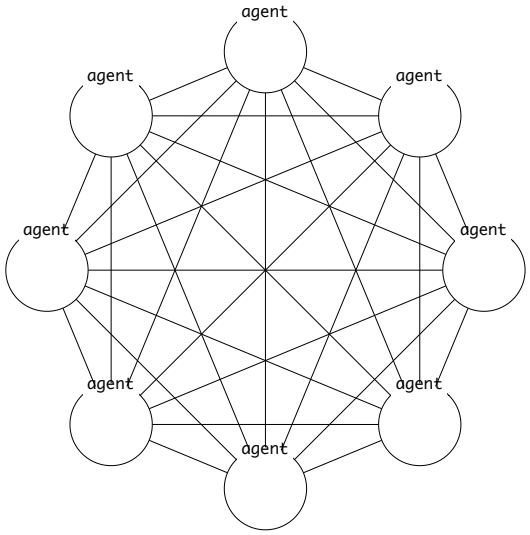


Figure 4.4: MUSCLE network of agents

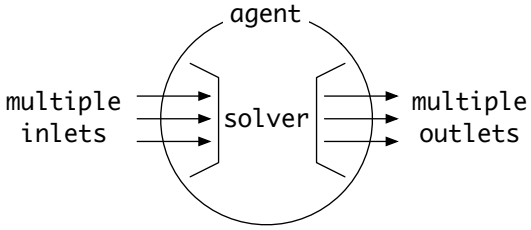


Figure 4.5: Wrapped solver with access to inlet/outlet slots of its agent

- introduce synchronisation points/barriers with a mechanism provided by the framework
- decide when to stop/finish the simulation

Yet, this is not the case for existing solvers, which have probably not been programmed with any coupling interaction in mind. These legacy solvers are considered to be the common case for a coupling scenario, which makes their easy integration a primary challenge.

MUSCLE agents play an important role when integrating a sub-solver into a coupled simulation. They act as a kind of mediator between the functionality of the framework and the logic of the raw sub-solver implementation (Figure 4.5). MUSCLE has been written in a way which puts as much of its functionality as possible into the core framework and the agent implementation, yielding only the necessary parts to the bridge between agent and solver code. This makes the interface between solver code and agent very "thin", which in turn leads to an easy integration of the solver code, as there are only very few relations between the solver code and the agent hull to implement.

At runtime, the solver logic will be called within the scope of the agent and thus has direct access to the functionality of the agent (i. e. member variables and methods). An empty sub-solver agent implementation has to be filled in three places to bring in the solver logic:

1. specify the temporal and spatial scale of the sub-solver, if appropriate
2. announce the inlet and outlet connections
3. execute the sub-solver (e. g. via calling its "main" function)

These three stages are further described in the following sections.

4.1.2 Coupling Relations and Message Flow

When a data message is being delivered, the start- and end-points of the connection are explicitly known. For coupling two participants, both have to take action. Within low-level network communication like TCP, the sending side opens a connection to the destination and the destination opens a connection to the sender. A handshake operation confirms the connection and data may be passed. The start- and end-points of the connection are identified with an IP address and port. A similar concept is used by the MPI standard, where the start- and end-points are specified with a "communicator" and a "rank". As with a port number, the rank is a simple integer number which might not be chosen freely: the port needs to be free on the corresponding machine and the MPI rank depends from the number of members of the communicator.

With low-level communication as TCP or MPI, the start- and end-points must be explicitly used within the code on both sides of a connection. Hence the coupling graph of multiple connected programmes is defined from *within* the code of every participating solvers. This approach is not practical if different existing solvers are to be coupled together. Once a

sub-solver is ready to be used with the coupling framework, it should not be necessary to alter its codebase just to define another coupling scenario.

This is why MUSCLE takes a special approach at how the coupling relations are defined. Here, the coupling relations are defined from the *outside* and not explicitly from within the individual solvers. When a sub-solver has data to send, it does not explicitly name the target sub-solver. Instead it passes the data to an "outlet", which is responsible for a special data content. This way the outlets are not directly associated with a specific destination sub-solver, but they are bound to a content of data. At each point where a sub-solver yields data, it does so via a dedicated outlet. The sub-solver connections to the outside (i. e. other sub-sovlers) can now be implemented independently, without the need to know the specific coupling scenario.

The resulting coupling-enabled sub-solver may be used with different groups of other sub-solvers, probably participating in various simulations. Likewise, a sub-solver of a coupled simulation setup might be replaced with another sub-solver, probably running an optimised codebase or being based on a different numerical model.

On the receiving side, the inlets work according to the same concept as the outlets do: a sub-solver receives a special content of data via inlets, but must not specify the origin (i. e. sending sub-solver). The outlets and inlets are defined and announced to the wrapping agent before the actual sub-solver execution occurs (section 4.1.1).

Looking from the outside, a sub-solver has several data consumers (inlets) and data providers (outlets). These have to be connected to couple the sub-solvers. Which pair of outlet/inlet has to be coupled is specified in the so-called *connection scheme* and is part of the configuration file belonging to a MUSCLE simulation.

Since each sub-solver probably uses a custom data format to store and interpret the data it passes to an outlet, it is unlikely that the receiving sub-solver is able to directly use (i. e. interpret) this data. For example, the sending sub-solver might store multiple three dimensional load vectors as single array of groups of x,y,z values, as shown in description 1. Whereas the receiving sub-solver might use another data format, like the one shown in

$$\{x_0, y_0, z_0,\ x_1, y_1, z_1,\ \ldots,\ x_n, y_n, z_n\}$$

Description 1: Example output format of the sending sub-solver

description 2, which is a common consecutive one dimensional representation of three dimensional array structures. Or maybe one sub-solver stores the data in a format where

$$\{x_0, x_1, \ldots, x_n,\ y_0, y_1, \ldots, y_n,\ z_0, z_1, \ldots, z_n\}$$

Description 2: Example output format of the receiving sub-solver

a special object is being used for each vector, to which pointers are stored in an array or linked list.

As it should not be required to alter the internal data representation of a sub-solver for a

coupling scenario, MUSCLE provides a mechanism to map the data from the sender to a format which could be used by the receiver. This is done outside of the sub-solvers in a so-called *conduit*. The conduit can intercept the data as it is being passed from an inlet to the connected outlet.

In order to connect the outlet data with the format from description 1 to a sub-solver which is expecting a data format at its inlet, like the one from description 2, a conduit can translate the data to the required format (Figure 4.6).

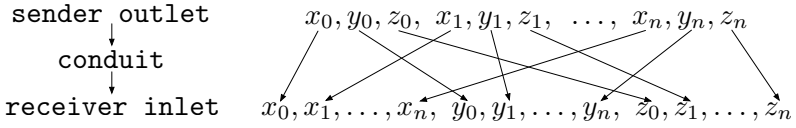


Figure 4.6: Mapping the load vectors from sender (description 1) to receiver format (description 2)

If a conduit is necessary to map the data for a connection, it is specified along with the outlet/inlet pairs within the configuration file for a simulation setup (Listing 4.6). Most simulation setups require custom conduit implementations. Thus, the conduits probably belong to a specific coupling scenario and their reuse is unlikely.

Nevertheless, the mapping implementation of a conduit often involves similar steps: reshaping of array data, multiplying or dividing with constant values, project 3D data to 2D or applying coordinate transformations. These general mappings may be grouped in so-called *filters*. MUSCLE provides a special kind of conduit, which acts as a container for these filters. Therein, an arbitrary number of filters can be applied to map the data to a format understood by the receiving sub-solver. Within the conduit, the data from the sending sub-solver is processed by each filter in succession and then passed to the receiving sub-solver (Figure 4.7). The filter mechanism is similar to the common *pipes and filters* architecture [19].

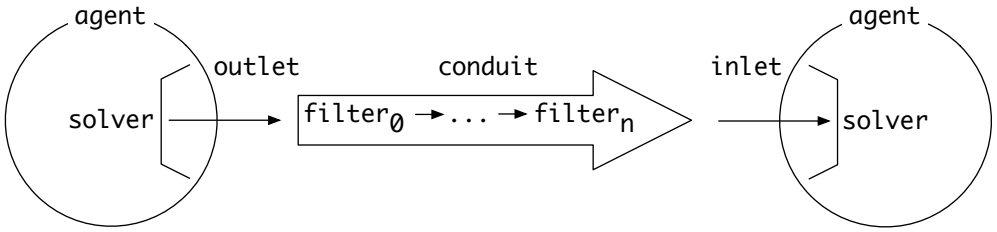


Figure 4.7: Mapping output data to input format using a chain of filters

The filter(s) for this conduit are specified within the connection scheme part of the configuration file. The setup system passes the filters to the conduit which dynamically builds its filter chain at runtime.

An example conduit to do the data mapping depicted in Figure 4.6 could use the implementation shown in Listing 4.1. This piece of code might be useful within other conduits as well. If put into a filter module, it can also be specified within the configuration file to be part of a filter chain to do more complex mappings. For example, it could be combined with a filter which does a projection from 3D to 2D, thus yielding an array of 2D load vectors to the receiving sub-solver.

This filter mechanism serves as a convenient mechanism to facilitate coupling scenarios more quickly. The concatenation of filters is not always an optimal solution regarding memory and CPU usage. If there are memory or speed constraints to consider within a conduit, it should probably be written as a dedicated implementation, thus creating a monolithic conduit which does not rely on any previously created, more general filters.

```
1  int vec_count = src.length/3;
2  double[] dst = new double[vec_count*3];
3
4  for(int i=0; i<vec_count; i++) {
5      // map x
6      int src_index = i*3;
7      int dst_index = i;
8      dst[dst_index] = src[src_index];
9      // map y
10     src_index = (i*3)+1;
11     dst_index = vec_count+i;
12     dst[dst_index] = src[src_index];
13     // map z
14     src_index = (i*3)+2;
15     dst_index = (vec_count*2)+i;
16     dst[dst_index] = src[src_index];
17 }
```

Listing 4.1: Exemplary algorithm to map the load vectors according to Figure 4.6

Along with the different data format for the outlet and inlet of a coupling connection, the size of the data may also change significantly during the mapping procedure. This is e. g. the case for 3D to 2D projections mentioned above, which would lead to strongly reduced data on the 2D end. A difference of factor 2 would result if the precision of floating point data has to be changed (e. g. going from double to single precision). Another case for very different data sizes exists if the output data from the sending sub-solver is being used to trigger an action on the receiver side. Here the conduit could scan the input data for its maximum value and pass a single true or false value depending on a threshold to the receiving sub-solver. For all these cases, it is usually preferred to only have to transfer the smaller amount of data over the network. For the 3D to 2D example, this would mean to do the mapping first and then transfer the smaller data via the network (Figure 4.8). To achieve this, the conduit can move itself to either side of the network. That is, the conduits executing object is being serialised and transferred to the machine where the designated sub-solver (i. e. the sending or receiving side) is located. The current

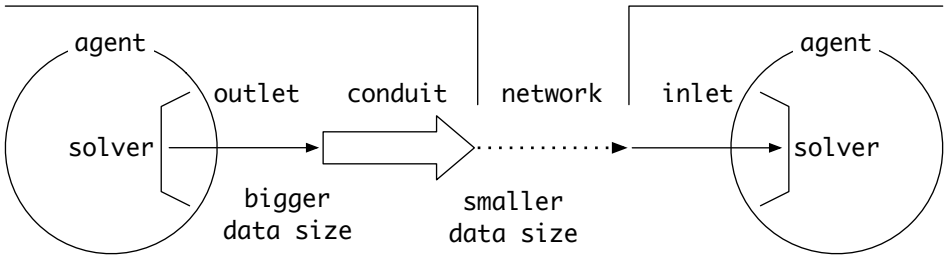


Figure 4.8: The conduit positions itself at the agent where the smaller amount of data has to be passed over the network connection

mechanism does move the conduit only once at initialisation, so as not to introduce any delays during the execution of the coupled simulation. This is generally sufficient, as the relation of data sizes between sender and receiver is implicitly given by the conduit mapping algorithm and does usually not change at runtime.

With many setups, the connected sub-solvers depend mutually on each other, e. g. the setup from section 4.4.1. That is, the dependency between a number of sub-solvers has a "circle structure" (Figure 4.9). Each of the sub-solvers could have an execution loop where

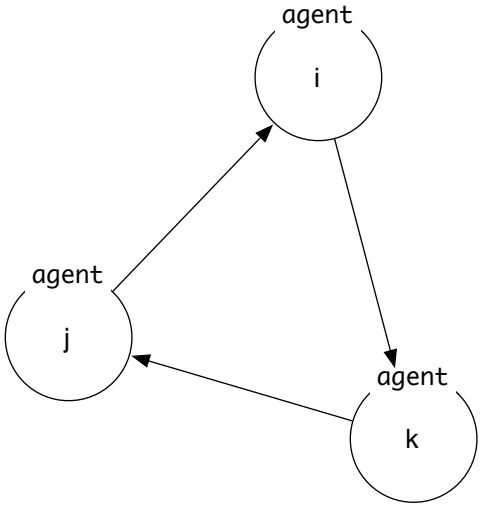


Figure 4.9: Dependency circle of connected sub-solvers (i, j, k)

it requires input data to be able to continue and process its output data. Such a main execution loop is shown in Listing 4.2

```
1  for(int step=0; !willStop(); step++)
2  {
3      // read from an inlet
4      inbuff = inletA.receive();
5
6      // use inbuff to calculate results for current step
7      // ...
8
9      // pass intermediate results to an outlet
10     outletA.send(outbuff);
11 }
```

Listing 4.2: Basic execution loop with output depending on previous input data

In order to run the simulation, one sub-solver has to provide an initial set of data, since it can not receive anything before its first iteration. For a single coupling setup this provides no problem, but if the very same sub-solver is to be used within another coupling setup, it might not be the first one in the circle and thus does not need to provide an initial data set. The message passing system in MUSCLE provides a solution for this dependency circle problem.

The coupling relations between the sub-solvers are not explicitly named from within the sub-solvers (see above). This way the receiving sub-solver does not have to choose itself from where the data is being sent. It receives the data on its inlet, as soon as there is data available. Now, it is possible to configure a second source for an inlet within the configuration file: one for the standard connection and one for the source of the initial data. The initial data can be send from a minimal "sub-solver" agent, so the configuration file defines two different sources for the inlet of sub-solver i: One from the initial data agent and the other from the sub-solver j (Figure 4.10). After the initial data has been sent, this agent terminates. On the following iterations, the inlet will be fed by the standard source sub-solver j. The active coupling setup will now be the one from Figure 4.9.

4.1.3 The Plumber Agent

After the simulation has been launched, the so-called "plumber" agent (`muscle.core.Plumber`) is responsible to configure the outlet/inlet connections. It relies on the connection scheme data to do so. The connection scheme can tell the start- and end-points for any of the configured coupling relations. The corresponding inlet for the calling outlet can thus be fetched from the connection scheme.

At simulation startup, the plumber agent gets hold of the connection scheme and first checks for possible deadlocks. This is a critical feature when creating and examining connection schemes. The current implementation for deadlock testing "unrolls" the predicted communication chain and checks if all requested data can actually be provided (for a given time) by the simulation setup.

Subsequently, the plumber agent creates (spawns) the conduit agents for each connection,

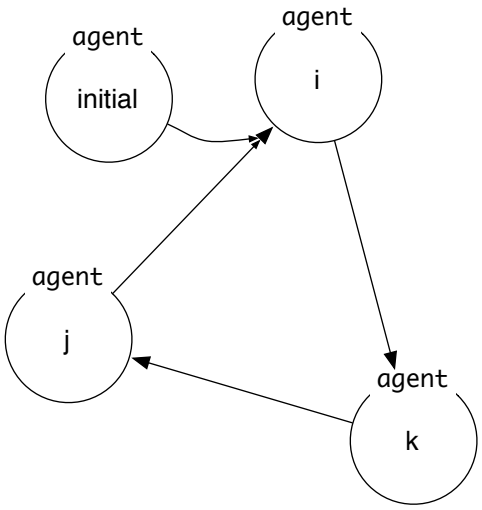
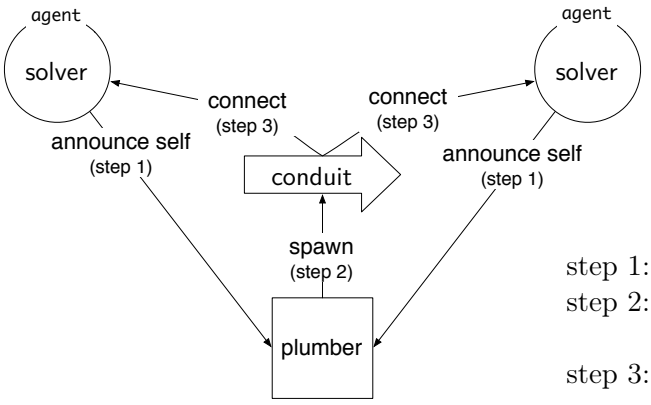


Figure 4.10: Dependency circle of connected sub-solvers with special agent to provide initial data

as soon as both participating agents have announced themselves at the plumber. This procedure is depicted in Figure 4.11.



step 1: required CAs are launched
step 2: new CAs announce themselves at the plumber
step 3: the plumber spawns designated conduit
step 4: conduit and the sink and source CAs form a peer to peer channel

Figure 4.11: Conduit spawning sequence

4.1.4 Agent Lifetime

In MUSCLE, every sub-solver is being executed under the hood of a software agent. This agent is an extended version of the standard JADE agent and thus is itself being executed under the hood of the JADE library. Three major stages are triggered for an agent under the control of the JADE library:

1. The agent is being instantiated via the JADE agent toolkit. This also involves a registration at the white pages agent of JADE. This is only possible, if the *name* of the agent does not already exist in the running platform. Otherwise the agent is terminated at this stage by the JADE library.
2. Next, the agent enters its setup stage. Herein all the MUSCLE related initialisation takes place (section 4.1.4.1).
3. After the setup is complete, the agent enters its main execution loop. In JADE, this stage is called the *agent life cycle* (section 4.1.4.2).

4.1.4.1 The Setup Stage

A MUSCLE agent can be configured with an argument list which is passed to the agent via the setup system from outside. The arguments define if the agent should either enter its normal execution procedure, or otherwise just print a description of its inlets and outlets and then quit. Any other arguments are forwarded to the argument list of the underlying sub-solver. This is similar to passing an argument list to the solver on the command line (i. e. the arguments passed to its `main` function).

After the arguments have been parsed, the agent will post a notification to its observers, passing information about the sub-solver (e. g. the name, inlet and outlet declarations). The observers can be used to add plugins to all agents, without altering any agent code. This way e. g. a graphical user interface can be added to receive information about all the agents or to log the state of sub-solvers in the simulation for debugging sessions. As a default, the agent does not use any observers, so no extra execution time is required here.

4.1.4.2 Main Life Cycle

As a last step, the agent enters its main life cycle. Within the life cycle, the agent is a fully activated member of the running JADE platform. The agent will not leave this stage until it is explicitly terminated. This is comparable to the main event loop of a standalone programme.

Herein the agent first communicates with the plumber agent, which is responsible to couple all the sub-solvers (section 4.1.3). Because all agents may be executed in parallel on different machines, it is possible that they are being launched before the plumber agent. Thus, the agents at this point will wait for the plumber to become available. Then they announce all their inlet and outlet declarations to the plumber.

Next, the agent tries to connect all its outlets to a destination. This is either a conduit leading to the appropriate sub-solver, or a direct connection to the sub-solver if there is no conduit required (section 4.1.2). As soon as all the outlets have been connected, the agent executes its encapsulated sub-solver. The agent does not need to wait for the inlets to be connected, as they are used in a blocking manner. That is, the sub-solver will need to wait anyway for the data to arrive at an inlet. This way the sub-solver can begin its calculation asynchronously, before all the other agents have finished their setup procedure. If the inlet required at a later point within the calculation (e. g. at the end of each iteration), this can reduce the duration of the overall coupled computation.

Before the sub-solver is started, the agent communicates with the white pages and the plumber and couples all outlets to their conduit (or directly to the particular sub-solver). All this interaction is only done once and does not interfere with the sub-solver execution (Figure 4.12).

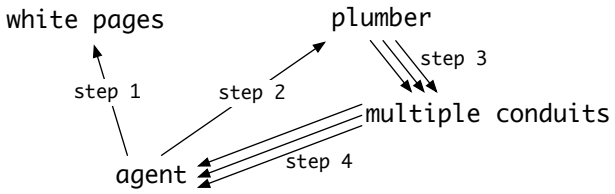


Figure 4.12: Communication carried out to integrate an agent to a coupled simulation

4.1.4.3 Message Delivery

During the computation, the communication between the coupled sub-solvers should be as fast as possible. Hence all the coupling setup is done only once at agent startup and thus does not slow down the execution of the sub-solvers at a later time. The outlet and conduits have all the information required to connect to the target. This is configured once at the coupling setup and then cached for fast access during the main calculation.

The following steps have to be addressed to deliver a data message to the receiving sub-solver:

MUSCLE 1. figure out the intended target agent of the message
JADE 2. determine its remote location
MUSCLE 3. determine to which inlet the data should be passed at the remote agent
MUSCLE 4. apply conduit mapping to the message data (if any)
MUSCLE 5. persist the message for network transfer (serializing/marshalling)
JADE 6. actually send the message across the network
JADE 7. receive the message and put it to its target agent
MUSCLE 8. extract the message content (deserializing/unmarshalling)
MUSCLE 9. apply conduit mapping to the message data (if any)
MUSCLE 10. put the message to the intended inlet slot

To carry out the message delivery, MUSCLE can delegate the steps 2, 6 and 7 to JADE. The agent lookup (step 2) is done by the JADE white pages (called *agent monitoring agent*, AMS), which keeps track of the location of the agents of a JADE platform. Steps 6 and 7 are carried out by the JADE internal message transport protocol (IMTP) [11].

As the core components of MUSCLE are written in Java, it is possible to serialize and deserialize complex data objects as well as transferring simple array buffers (steps 5 and 8). With MUSCLE this is even possible if there is no default serialisation mechanism for a data structure (i. e. not all objects implement the `java.io.Serializable` or `java.io.Externalizable` interface), MUSCLE provides a fallback mechanism: the stream serializer XStream [176] is being used to transfer the data for these rare cases.

The other core components of MUSCLE are responsible to carry out steps 1, 3, 4, 9 and 10: For steps 1 and 3 , the plumber is required, who determines which outlet from an agent should be connected to which inlet at the receiving agent. If the connection uses a conduit, it is responsible for applying the filters to the message data, steps 4 and 9 (section 4.1.2).

The steps responsible to determine the path of the message to its target (steps 1, 2, 3, and 10) are preconfigured during the setup stage, before the actual sub-solver execution starts. They are cached within the outlets, without any further lookup required.

Next to the message delivery mechanism, MUSCLE provides a way to notify the sender of a message if the receiver can not allocate sufficient memory to hold the message. This might be the case, as the connected models may have different purposes and different memory requirements. In such a case, the receiver can call back to the sender and post a notification to tell when sufficient memory is available. The sender now re-sends the message. This happens internally in the MUSCLE message delivery and does not have to be implemented by the sub-model developers.

4.1.5 Multiscale Operation

The sub-solvers in a coupled MUSCLE simulation may be dedicated to solve different numerical problems. Keeping the scales separated will actually result in an overall computational speedup compared to a (monolithic) model coupled at the finest scale.

When a sub-solver is wrapped into a software agent to be able to join a MUSCLE simulation, it is usually configured to allow for participation in different simulation setups. Any requirements which are special for the current simulation setup are implemented in the conduit functionality and configured via the parameters passed from the configuration file (section 4.1.2).

While data is being passed from a sub-solver outlet to the inlet at the receiving sub-solver, the format of the data may be adapted using a conduit, which includes the mapping logic, often specific for an individual coupling scenario. Thus, the sub-solvers carry out the numerical computation and the conduits are responsible for the coupling interactions.

The more the conduits know about the outlet/inlet which they connect, the better they can ensure that the connection scheme is valid. The only mandatory information at both ends of the conduit is the particular data format. Now, the passed outlet/inlet data is usually custom for each sub-solver. But the particular sub-solver usually belongs to a spatial and temporal scale it operates on. This can be used as information of the conduit, to tell if the connected outlet/inlet can be assigned in terms of the scales at both ends. This way, the conduit has two kinds of information to verify a coupling interaction: The data types at both ends must match, and the scale information must be compatible too.

The description of the scales is assigned to a sub-solver using SI units and it is directly expressed within the implementation of a sub-solver agent. This way, the scale information is not only readable by the programmers, but can be evaluated at runtime, thus being available to MUSCLE, too. To express and compare the SI units within the programme, an SI units library has been utilised [97].

Usually, the inlets/outlets directly inherit the scale description from their host solver. Otherwise, a separate scale description might be added to an outlet or inlet. This information can now be used for two purposes.

During conduit/filter implementation

When implementing a mapping procedure, the conduit (or filter) can be written for more general cases, as they could use the scale information to take care of their mapping, e. g. from coarse to fine temporal scales (i. e. via interpolation) without the need to know the precise scales within their code. This could even be done without explicitly configuring the filter from the outside, as it can determine the ratio of the two time scales according to the scale description at both ends.

At the sub-solver start

The scale information is also being used at the setup stage. Here the conduit will check if the scales at both ends are compatible. As the scales of the connected sub-solvers may be different, the conduit has to know how the scale information changes during the mapping of the data.

If the data is mapped directly from outlet to inlet format within a custom conduit implementation, the conduit does know its own mapping procedure and if it can match the incoming scale to the outgoing one. If the conduit itself consists of a chain of filter modules (section 4.1.2), it has to gather the scale of its incoming and outgoing ends from the individual filters. In order to do so, the filters can expose the scale of their incoming side. This is usually computed during the initialisation of the filter, depending on the required output scale of the filter. Thus, the filter applies a general mapping procedure to the scale description itself. This allows to evaluate the cumulative scale mapping of all combined filters. As this mapping is applied at runtime, this also works for an arbitrary combination of filters.

Now the conduit can apply all the scale mappings from one end, via all the filters, and

see if the result matches the one for the other end-point. This validation is only done once while the conduit connects to its both sub-solvers and so does not slow down the message passing during computation.

Next to the mapping of scale information, the data type at both end of the filters is evaluated for the two end-points of the conduit. This is also being used to determine if the coupling connections are valid. To enable this preflight check also for the sub-solver connected directly, i. e. without requiring a conduit mapping, a temporary conduit is being used at launch time to connect these two sub-solvers. It validates the scale and data type information and terminates itself.

In case of incompatible couplings, the MUSCLE framework will follow the fail fast principle [151]. As such, a simulation will not run if either the scale or data type of any of the connections can not be mapped between outlet and inlet and will immediately bail out.

4.2 The MUSCLE Setup System

A MUSCLE simulation has the ability to integrate different kinds of sub-solvers which may run on distributed hardware systems. These connected machines may feature heterogeneous hardware and operating systems. Due to all these interacting aspects, the configuration of the whole coupled setup and also the actual launching of the individual sub-solvers become a significant topic to be considered.

Configuring a coupled model has three important setup stages: first, the necessary sub-solvers have to be prepared and glued together.

Then second, the setup phase where a specific simulation scenario will be configured: providing a geometry, boundary conditions and other input parameters, defining a termination criterion and choosing how the results should be dumped for visualisation. This kind of setup also has to deal with the runtime infrastructure where the simulation will actually be processed. For distributed HPC systems like large scale compute clusters it usually makes a difference if data should be dumped locally or to a shared file system.

The third setup phase addresses actually running a simulation. The programme has to be configured individually for different distributed hardware environments or operating systems (homogeneous/hybrid, slow or fast network, shared global or separated local file system). Furthermore, the programme has to be launched, i. e. deploy the individual jobs to the available machines. This last step, where the simulation is actually deployed and executed, is often very different depending on the given HPC environment, most of which use batch queueing systems, requiring specific configurations.

A common approach is to use some kind of configuration file, which expresses the current setup and is able to pass additional parameters to the programme. Configuration files are often written as plain text files subsequently being parsed by the programme at startup. This easy to maintain interface to the programme is suitable to be human writable and readable. It should be possible for everybody who actually *uses* the programme to write

this configuration file. This includes also people who are not involved in the development of the programme and do not necessarily know its source code. The configuration file is the bridge between user and programme.

For a MUSCLE simulation, there are different aspects related to the setup of a coupled simulation. This requires a configuration mechanism allowing to express all these topics in a manner also suitable for non-programmers.

1. The most important configuration part is the declaration of the involved sub-solvers together with their coupling interactions (connection scheme). This is specified in a configuration file specific for each coupling scenario.
2. The aforementioned configuration file also contains custom settings for the coupling setup, as e. g. the computational domain or termination criteria (e. g. number of time steps).
3. Similar to the above settings, the individual sub-solvers may need to know properties related to this very simulation setup, like the proper temporal and spatial scales, domain settings or discretisation levels. Hardware resources like a minimum stack size required for special algorithms may also be configured here. These settings are also specified in the configuration file, together with 1 and 2.
4. Usually the sub-solvers can also be used as standalone programmes. As such, they probably require special configuration settings, which do not depend on the context of the overall coupling setup. These are configured as the individual sub-solver requires, probably in a separate configuration file or via environment settings. This involves the setting of output directories and dump frequency or switching between execution modes like debug and release or setting the logging verbosity.
5. In addition to configuration aspect (4), the coupling framework also has configuration properties which are independent from the current coupling scenario (selecting machine dependent library paths, debug/release modes or assertion activation). This can be configured in different places: There is a default configuration which is suitable for most machines. This may be extended or replaced by machine specific settings. Also, every user may add or alter these setting with a custom settings file. MUSCLE also allows to define these settings ad hoc via command line options.
6. Other configuration parameters allow the user to define on which machines the individual sub-solvers (and the plumber and white pages) should be launched. The default behaviour is to spread the sub-solvers evenly to the available machines. If some sub-solvers should be grouped together or launched on specific hardware, this can be configured via the command line interface. Here one can also specify if a machine may only be reached via specific ports (e. g. because of firewall restrictions).

In order to configure all these setting in a maintainable manner, a custom setup system has been developed. This takes all the user defined configuration properties and command line arguments, determines operating system specific paths, looks for hardware resources like available memory and number of CPUs and subsequently launches the distributed

simulation with the appropriate settings for each sub-solver and machine (Figure 4.13).

Figure 4.13: Launching a simulation via the setup system

For the implementation of the setup system, the Ruby programming language has been used [165, 145] which allows to interact with the operating system and file systems in a flexible *platform independent* manner. Thus, the configuration mechanism is the same for every machine. Because the major parts of the setup are specified in the configuration file for each coupling scenario, this file can be written in a "mini programming language", specially designed for the coupling framework requirements. These kind of languages are also called domain specific languages (DSL) [57, 65]. Within the configuration file, all standard Ruby programming constructs may also be used next to the DSL syntax for maximum flexibility (Listing 4.8). These mixed style DSLs are called *internal DSL* ([57, 65]).

Even though the setup system has been implemented in Ruby, MUSCLE can be used without a Ruby version installed on the target machines. The setup system can be executed in Java along with the MUSCLE core library, which is possible due to the jruby library [135, 85].

Within the configuration file, the sub-solvers taking part in a simulation are declared as shown in Listing 4.3. In this example, two sub-solvers are declared using the `add_solver` statement. The last item of the declaration specifies the class name of the corresponding agent implementation (e. g. `solvers.Wind`), to which an alias name is being assigned (`wind`). This name is used within the connection scheme declaration to refer to this sub-solver. At runtime, the agent for this sub-solver is registered at the white pages using this name.

```
1 add_solver "wind", "solvers.Wind"
2 add_solver "bridge", "solvers.RigidBridge"
```

Listing 4.3: Agent declaration in configuration file

The connection scheme for this setup can now be declared (Listing 4.4).

```
3 attach "wind" => "bridge" do
4   tie "load_vectors"
5 end
6
7 attach "bridge" => "wind" do
8   tie "surface"
9 end
```

Listing 4.4: Connection scheme declaration in configuration file

The connections from one sub-solver to another sub-solver are specified within an `attach` block. Herein all the outlets and inlets of the *same* pair of sub-solvers are specified for one direction. Each outlet to inlet connection is declared using a so-called `tie` statement.

As the corresponding outlet and inlet names usually have different names assigned within their respective agents, the `tie` statement usually has to know two different names, one for the outlet and one for the receiving inlet (Listing 4.5). As the `tie` statements are grouped within the `attach` block, it is always known to which sub-solver the outlets and inlets belong.

```
3 attach "wind" => "bridge" do
4   tie "load_vectors", "load"
5 end
6
7 attach "bridge" => "wind" do
8   tie "road_surface", "obstacle_boundary"
9 end
```

Listing 4.5: Connection scheme association differently named outlets and inlets, i. e. `road_surface` to `obstacle_boundary` (variant of Listing 4.4)

If there is a conduit required to map the outlet data to a format intelligible by the inlet, it can also be specified using the `tie` statement (Listing 4.6). Similar to the declaration of the sub-solvers, this directly refers to the class name of the conduit. In line 4, a custom conduit implementation is assigned to the connection. Line 8 uses a conduit provided by MUSCLE, which can automatically load a filter chain (section 4.1.2). Here, a filter is loaded which multiplies the output values with a constant factor, e. g. to convert from kilo to milli.

```
3 attach "wind" => "bridge" do
4   tie "load_vectors", "load", "conduits.VectorMapper"
5 end
6
7 attach "bridge" => "wind" do
8   tie "road_surface", "obstacle_boundary", "muscle.core.conduit.
      AutomaticConduit", "filters.multiply(10**6)"
9 end
```

Listing 4.6: Connection scheme using conduits (variant of Listing 4.4)

Additional properties for the simulation are being set to a key value pair storage, which later at runtime is available to all the distributed sub-solvers (Listing 4.7).

```
10 env["timesteps"] = 42000
11 env["nx"] = 200/2
12 env["ny"] = 50
13 env["nz"] = 50
14 env["dx"] = 2
```

Listing 4.7: Simulation properties in configuration file

This storage automatically distinguishes between strings, integers and floating point types and may use relative file paths or do calculations on numbers within the configuration file. Most of this is possible due to the underlying Ruby language. This simplifies the setting of specific properties depending on the machine name (or IP address) or the current date and time.

A common task with running numerical simulations is to do parameter studies, where a simulation setup is being run multiple times and a specific parameter (or set of parameters) is repeatedly adjusted. As the simulations are not identical, this requires separate configuration files for each execution. The problem with this procedure arises, if a setting which is identical for all configuration files has to be changed (e. g. a different number of time steps or using another conduit). This requires changes in all the configuration files of the parameter study. Also, it is not immediately obvious, where the individual configuration files really differ, i. e. what parameters are being subject to the parameter study. The MUSCLE configuration files provide a solution to this issue: They can be inherited from a basis configuration file and only change some parameters or introduce completely new settings. Listing 4.8 shows a minimum example configuration file (`Setup2.txt`), which inherits from another file (`Setup1.txt`).

```
1 # file Setup2.txt
2 load(File.dirname(__FILE__)+"/Setup1.txt")
3 env["nz"] = 100
```

Listing 4.8: Inherited simulation setup with slightly changed properties

This is possible using a single line of Ruby code (line 2 in Listing 4.8). Herein, the `load` call is being used to include the contents of another file. This file is given with an absolute file path, which is crucial to make the configuration file machine and user independent. This path is being constructed from the absolute path to the current configuration file (`Setup2.txt`) using the `__FILE__` specifier. From this, the path to the containing directory is evaluated using the Ruby command `File.dirname`. The name of the base configuration file (`Setup1.txt`) is now appended to this directory path and consequently the parameter specific for this setup is changed (line 3 in Listing 4.8). This leads to a slightly different, easy to maintain configuration file for parameter studies. Changes in the base file will automatically be considered in all descendant configuration files.

The setup system can also be used to directly launch the individual sub-solvers on the

machines of a distributed system. As the rest of the setup system, this launch functionality has been implemented using Ruby. The same mechanism is also being used to launch distributed programmes with the Bond framework (section 5).

The different MPI implementations bring a similar tool to launch an application on multiple hosts, which is called `mpiexec` within the MPI standard. The two commonly used MPI libraries, MPICH2 and OpenMPI, use different solutions. With MPICH2 this is done using a python script [56], whereas OpenMPI uses a library called Open Run-Time Environment (orte) [24].

Figure 4.14 shows a comparison for the time it takes to launch a simple programme on up to 40 machines. For $\mathcal{O}(10)$ machines, the overhead due to the used launch procedure is usually negligible, but may become an issue on clusters of larger scale. It has to be noted, that some implementations of the `mpiexec` launch mechanisms can directly interact with a queueing system, if available, and use its task manager to speed up the launch of distributed programmes.

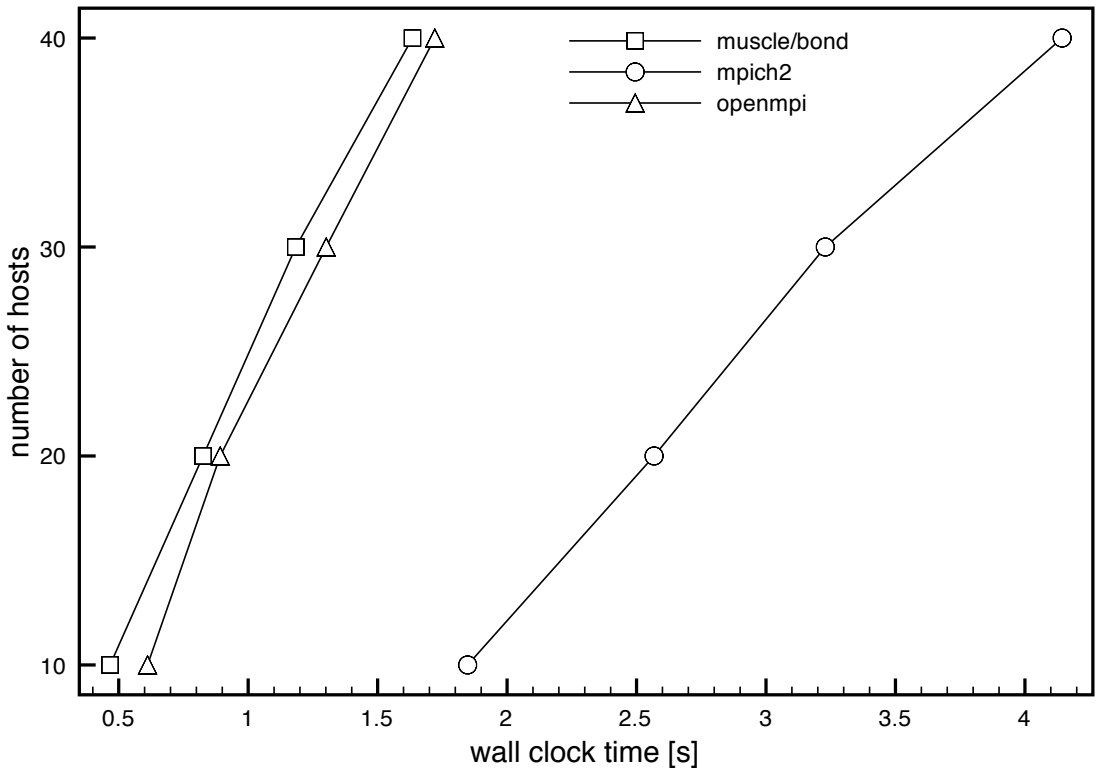


Figure 4.14: Comparison of the launching time for parallel jobs using MUSCLE/Bond, MPICH2 and OpenMPI

4.3 The MUSCLE Native Library

The core library implementation of MUSCLE has been implemented using the Java language (see section 4.1). This is due to the fact that MUSCLE used the JADE agent framework internally. As JADE is a pure Java implementation, Java became the language of choice for MUSCLE. This way MUSCLE can integrate the JADE functionality quite naturally.

To maximise the number of potential legacy solvers to build a coupled scenario, MUSCLE cannot limit itself to Java-only sub-solver implementations. Thus, the other important programming languages of HPC computing, C++, Fortran and C, have to be supported as well. These three languages are especially popular within the HPC community, because the MPI standard provides language bindings for only these three programming languages [125].

Compiled Java source code results in a portable intermediate format, the bytecode. This is then to be interpreted by a platform specific runtime environment, the Java Virtual Machine (JVM) (section 3.4). The bytecode itself is thus platform independent, or *non native*. On the other hand, platform dependent binaries can be called *native*, i. e. they depend on a specific hardware architecture and operating system. Hence, programming languages which result in non portable binaries (more specific: object code), like C++, Fortran or C, are usually coined as *native languages*. Languages resulting in portable bytecode are e. g. Java and C# [72, 156].

In vanilla Java, native languages can communicate with the "Java-world" using the Java Native Interface (JNI) (section 4.3.1). The default JNI features a very low level C API which is a great step away from the modular approach used within MUSCLE. After all, creating an agent from native (legacy) code and thus providing its functionality for a coupled simulation should be a straight forward process.

4.3.1 The Java Native Interface

A part of the Java language is the Java Native Interface (JNI). It specifies an API which can be used to interact with the JVM from within C programmes [111, 38]. This can also be used the other way around, i. e. to call native functions from within Java code.

The JNI provides a very low level access to Java from a C interface, so one might call it a "C to Java bridge" (and vice versa). A simple binding API for C++ exists, but it merely simplifies calling the C API from C++ code in a manner which better fits the C++ syntax. The C++ API has the benefit of being more type-safe for the JNI reference types (Table A.1), which the compiler can not distinguish when using the JNI C API. Apart from this there is no difference in functionality, the programming idiom is still the same as for the C API (this is comparable to e. g. the C++ bindings for MPI). There are no bindings for other native languages like Fortran.

With mixed Java and native programme parts, the code which issues the first call via the JNI bridge can be either the Java or native C/C++ side. But if the native code wants to initiate the communication, the whole Java runtime has to be launched from within the C/C++ code.

For the native interface of the coupling framework, the C++ bindings for the JNI C API have been used because they are more clean to use and less error prone because of the introduced compile time type checking.

The JNI distinguishes two major data types: value types and reference types. The former refers to primitive data types (`int`, `double`, `char` etc.), whereas the latter ones represent a handle to a Java object. The value types are directly copied from Java to C and vice versa. For example passing a Java `int` type as an argument to a C/C++ function will yield a `jint` within the body of the function implementation on the C/C++ side. The usage of value types is straight forward with the traditional JNI.

When dealing with reference types (i. e. Java objects), this simplicity no longer exists. An opaque reference is passed to the native code on which one has to carefully apply the accompanying JNI function calls to get hold of the data from the native side. Here one also has to ensure to retain the data, so the garbage collector (GC) will not be able to destroy it. This is even the case for the most common kind of reference objects: arrays of (identical) primitive data types, like an array of double precision values (`double[]`). The major reason for this are the different concepts which with the JVM treats its heap and the heap of native applications, or rather the fact that there are two different views on the heap at all.

4.3.2 Within the Native Library

The MUSCLE support for native solvers consists of several components and methodologies, which make the integration of native agents a straight forward process. Together they compose the MUSCLE native library (Figure 4.15).

Basically, the MUSCLE native library has been modelled to allow access to all the functionality of the MUSCLE Java agent also for native sub-solvers. To do so, all software components which are directly used from a sub-solver (e. g. the outlet/inlet declarations, using the outlets/inlets, access to the simulation properties or the stop condition) have a counterpart implementation written in C++.

These counterparts directly bring the OOP based functionality of the MUSCLE software agents to native C++ code. Unfortunately, the functionality of the JNI was not sufficient to directly bridge the interface of the counterparts to its corresponding implementations on the Java side. As a solution, several software modules have been written to allow high level OOP concepts between the Java and C++ (native) code.

Contrary to C++/C/Fortran, Java features a garbage collector to automatically free memory regions which are no longer used by the programme (section 3.4). With C++/C/Fortran,

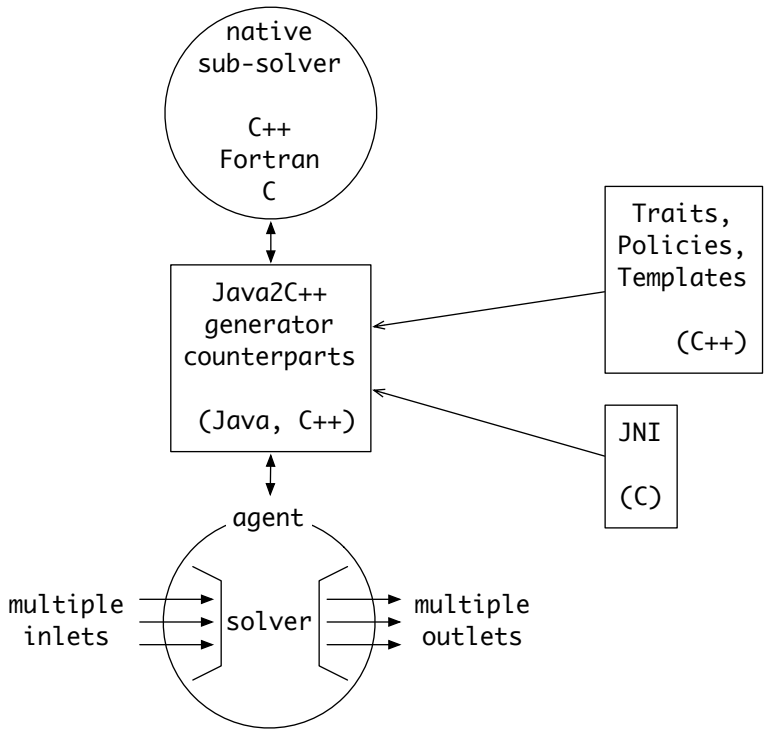


Figure 4.15: Components and methodologies of the native library

the programmer explicitly has to take care of memory management. These different concepts have to be fused by the MUSCLE native library, allowing to share data from the Java code with data on the C++/C/Fortran side (and vice versa). For this shared data, MUSCLE has to ensure that it is not inadvertently being released while native code is still using it. This is all done from within the MUSCLE native library and doe not require special consideration within the sub-solver implementation.

Within the JNI, a set of procedures exist to read and manipulate data (i. e. data stored in arrays or single values). Another type of functions is available to call Java methods, with data values being passed in the argument list and data received as the result of the method call (i. e. return value). Together, there are about 18 different possibilities (Table 4.1). The problem with the JNI is, that the data types being used in these functions is part of

`CallDoubleMethod`	`GetDoubleArrayElements`
`CallDoubleMethodA`	`GetDoubleArrayRegion`
`CallDoubleMethodV`	`GetDoubleField`
`CallNonvirtualDoubleMethod`	`GetStaticDoubleField`
`CallNonvirtualDoubleMethodA`	`NewDoubleArray`
`CallNonvirtualDoubleMethodV`	`ReleaseDoubleArrayElements`
`CallStaticDoubleMethod`	`SetDoubleArrayRegion`
`CallStaticDoubleMethodA`	`SetDoubleField`
`CallStaticDoubleMethodV`	`SetStaticDoubleField`

Table 4.1: JNI functions where the processed data type is encoded in the function name (here: double type)

the function name, e. g. `CallDoubleMethod`. Now for most of these functions there are 10 different versions, one for each available data type, including `void` (Table 4.2). The others have 8 versions, as the `Object` and `void` versions are not always appropriate. Access to all

`CallBooleanMethod`	`CallIntMethod`
`CallByteMethod`	`CallLongMethod`
`CallCharMethod`	`CallObjectMethod`
`CallDoubleMethod`	`CallShortMethod`
`CallFloatMethod`	`CallVoidMethod`

Table 4.2: Versions for the JNI function `Call...Method`

these functions has been unified with a general concept. Herein, only the purpose is part of the function name (e. g. `CallMethod`) and the corresponding JNI function is automatically being selected at compile time depending on the data type within the argument list. Thus, there is only one function required to e. g. pass data to an outlet. Otherwise there would have to be different versions for all the possible data types. The implementation uses the programming technique of *traits and policies* together with the C++ template system

to implement this generic access to the JNI [2, 169]. It effectively creates the versions required for all the method and type combinations during the template expansion stage of the build process. Hence, there is no performance loss at runtime, but the codebase of the MUSCLE native library and also the coupled sub-solvers is much easier to maintain.

4.4 MUSCLE Use Cases

The following section describes coupled systems which have been realised using MUSCLE. They focus on the ability to couple different numerical solvers easily. These coupling scenarios may also be called complex automata (section 2.1).

A number of performance tests, such as an evaluation about the systems scalability has been performed together with the developed Bond framework (section 5).

4.4.1 Sediment Transport and Erosion Process with Flow Solver

The parallel fluid solver from section 3.3.2 has been the first non trivial use case for a coupling setup. Although this solver uses a variable number of processes, all of them use the same internal solver. They merely differ at the boundary of the individual sub-domains.

So for a multi model use case for MUSCLE, a coupled application should integrate solvers, where the coupled computational models are different. This coupling scenario should also allow to verify the functionality of the coupling framework during its development. To make this feasible, the coupling setup has been created from an existing solver which addresses multiple simulation elements. The existing solver implements a model to simulate river channel erosion and sediment transport within a fluid flow [120, 47, 32].

The work done on this sediment-erosion-model is part of another project [157] and is written in the C++ programming language. The solver uses a modified Lattice Boltzmann automaton (section 3.3.2.1) to simulate incompressible flow where terms to simulate buoyancy were added in combination with an Lattice Boltzmann based solver for advection diffusion of sediment. It is capable of using a non-uniform lattice as the computational domain.

The main tasks of the sediment transport and erosion process within a fluid solver are the following:

deposition During the deposition phase, the sediment particles fall down and stop moving. They are now considered to belong to the solidified bottom boundary and present an obstacle for the fluid.

erosion While erosion occurs, sediment particles on the bedrock are lifted into the fluid and can be carried away depending on the local shear-stress.

toppling The process of local piles of sediment which fall (topple) into adjacent lower areas (holes) is called toppling. This effect also occurs at very steep slopes.

fluid movement The movement of the fluid is the driving force which transports the floating sediment and is obtained from the Lattice-Boltzmann solution.

advection Some of the sediment particles are suspended within the fluid. They are moved due to the local velocity of the fluid.

diffusion The suspended sediment particles in the fluid move along concentration gradients. This diffusion is calculated together with the advection.

sediment concentration The suspended sediment has a varying concentration in the fluid, which is calculated separately from the flow field.

Albeit the model does solve multiple problems internally, the legacy code basis is a monolithic entity.

The configured setup uses a 2D domain with a coarse discretisation for the bulk flow areas, which becomes finer when approaching the interface between the fluid and the original sediment layer (Figure 4.16). The sediment at the bottom of the domain is changing its shape due to the erosion/deposition/toppling processes.

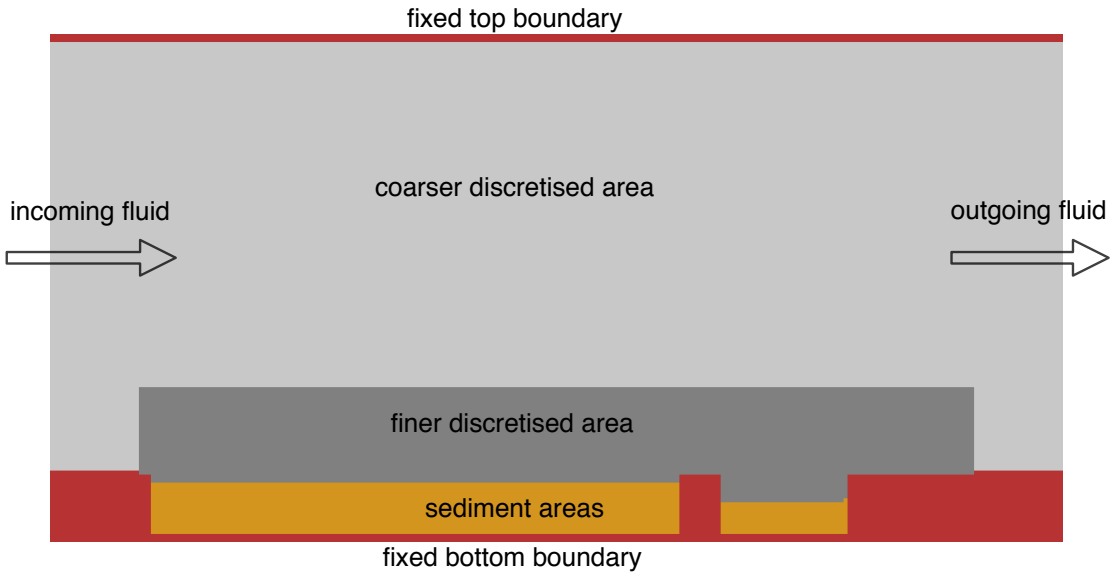
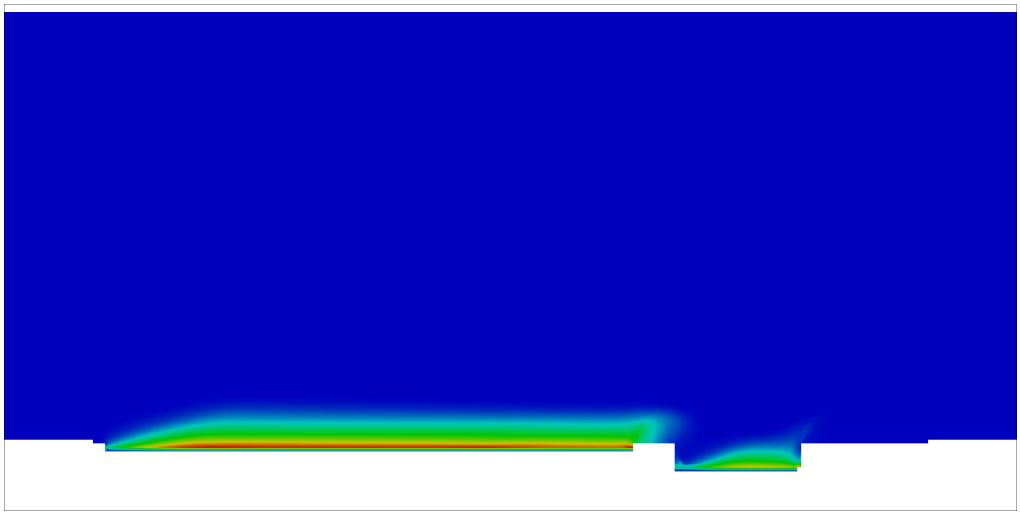


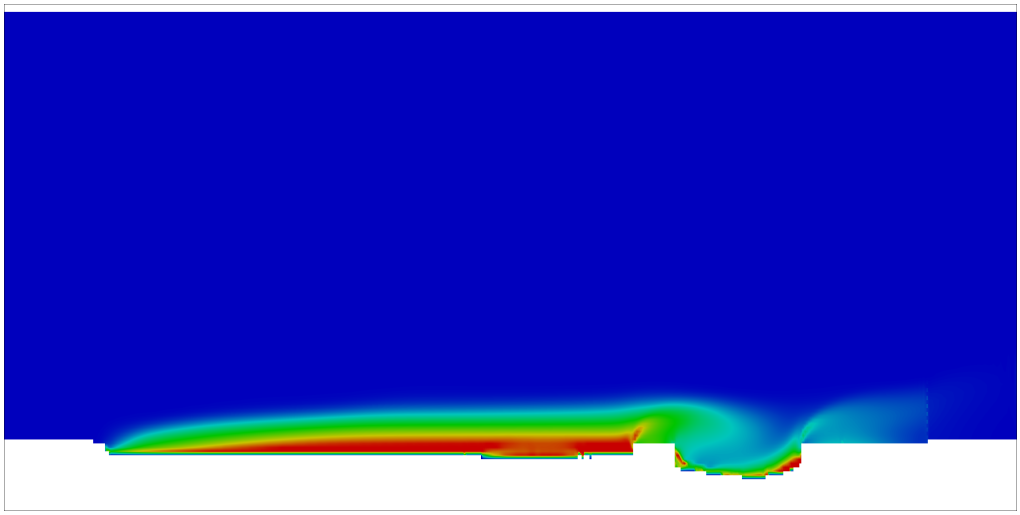
Figure 4.16: Simulation domain for sediment transport problem on non-uniformly discretised grid

Figure 4.17 shows two different snapshots of such a simulation where the current sediment concentration is displayed: Some parts of the bedrock are subject to erosion (removal) of sediment, where in other areas deposition (adding of sediment) takes place.

Three mutually interacting solvers where developed from the original monolithic code base:



time $t = 500\,\Delta t$



time $t = 1000\,\Delta t$

Figure 4.17: Sediment concentration (light blue, low concentration to red, high concentration) and boundary changes (white line) due to erosion.

Sedimentation Solver

A sedimentation solver to simulate the sediment deposition, erosion and toppling processes has been separated from the monolithic code base (Listing 4.9). To be able to apply these three stages for the sediment particles, the sub-solver needs to know the *sediment concentration* near the interface between fluid and non-movable sediment vertices (line 4). The interface is only changed within this sub-solver, hence it is not exposed to the other sub-solvers.

To be able to proceed with the calculation the mandatory input required for this sub-solver is the concentration of the sediment in the fluid. It is computed by the advection diffusion sub-solver (Listing 4.11, line 17).

All the three stages within this sub-model, deposition, erosion and toppling, may change the sediment concentration and also the boundary state of a sediment vertex (lines 7 to 9). This switch determines if a vertex of the domain lattice should be associated with the boundary. After a calculation step, these flags are passed as Boolean values to an outlet. To further describe the interface to the solid sediment vertices, the normalised distances to the sediment is being passed as double precision values.

In the coupled simulation, this information about the changed sediment boundary is then received by the fluid sub-solver.

```
1  for( t=0; !solver->stop(); t++)
2  {
3      // receive sediment concentration
4      JNIArray<jdouble>& cs = c->receive(cs);          sediment concentration ⟸
5
6      // calculate sediment concentration changes and change boundary
           accordingly
7      deposition(...);
8      erosion(...);
9      toppling(...);
10
11     // send sediment boundary changes
12     activesWriter->send(actives);                    sediment boundary ⟹
13
14     qsWriter->send(qs);                              sediment boundary ⟹
15 }
```

Listing 4.9: Sedimentation solver execution model, shown in pseudo code

Fluid Solver

The fluid flow is calculated with a Lattice Boltzmann based solver which can operate on a non-uniform lattice (Listing 4.10 and section 3.3.2.1). Due to the different levels of discretisation, the model uses a nested time stepping scheme which executes two step on the finer grid for one time step on the coarser lattice. The Lattice Boltzmann discretisation uses a D2Q9 stencil, i. e. 9 discrete microscopic velocities on a two dimensional lattice.

Within the computation domain, the fluid solver operates on all vertices which are "movable", i. e. which do not belong to the parts occupied by solidified sediment (and the limiting top boundary). So the only thing that is subject to change outside the control of the fluid solver is the changing sediment boundary. This is being read by an inlet before each calculation step (lines 4 and 6). Thereafter, the solver transfers changes across the interface between the differently discretised grid areas, from coarser to finer resolution (line 9).

Now the interaction between the microscopic particle distributions of the fluid model are computed (line 12). This uses information which is available locally at each vertex to calculate the new discrete distributions. Next, the microscopic particle distributions are moved along their 9 discrete directions. This is being calculated in the propagation step (line 13).

As the last step of the fluid simulation, the data buffer holding the collision results is being put to a temporary storage (line 14). This ensures that the newly calculated collision results within the next iteration do not interfere with the old results, which are required to compute the new ones.

At the end of each iteration, the fluid solver uses a MUSCLE outlet to send the fluid velocities (line 17). These are kept in the format used by the monolithic solver (i. e. particle distributions) and are not translated to SI units each time. As the sub-solvers use the same data formats internally, the receiving side would otherwise need to undo the SI unit representation which would result in two unnecessary computations for each communication.

```
1  for( t=0; !solver->stop(); t++)
2  {
3      // receive sediment boundary changes
4      activesReader->receive(actives);                    sediment boundary ⟸
5
6      JNIArray<jdouble>& qs = qsReader->receive(qs);      sediment boundary ⟸
7
8      // transition between coarse and fine grid level
9      scale(...);
10
11     // calculate flow solver results
12     collision(...);
13     propagate(...);
14     setFs(...);
15
16     // send velocity
17     velocityWriter->send(velocity);                     fluid velocity ⟹
18 }
```

Listing 4.10: Flow solver execution model, shown in pseudo code

Advection Diffusion Solver

The simulation of the sediment movement within the fluid has been extracted to another separate sub-solver (Listing 4.11). As within the monolithic code base, it uses a separate Lattice Boltzmann model to compute the movement and spreading of the sediment within the fluid (advection and diffusion), which is described by the following advection-diffusion equation.

$$\frac{\partial s}{\partial t} + \vec{u}\,\nabla s = D\nabla^2 s \tag{4.1}$$

With s describing the sediment concentration, $\vec{u}(\vec{x},t)$ being the velocity field and $D(\vec{x},t)$ the diffusion coefficient [157].

The lattice of this model uses the same non-uniform discretisation as the fluid solver, but operates with only 5 microscopic velocity directions (D2Q5).

The driving force for this solver is the current fluid velocity. This is obtained via an inlet before the calculation starts (line 5). As this solver is also based on a Lattice Boltzmann model, the following steps are similar to the fluid solver. Thereafter, the current sediment concentration is being computed from the Lattice Boltzmann distributions calculated in the previous steps (line 14). This concentration is now being passed to a MUSCLE outlet (line 17).

```
1  for( t=0; !solver->stop(); t++)
2  {
3      // receive velocity
4      JNIArray<jdouble>& velocity =
5              velocityReader->receive(velocity);          fluid velocity ⟸
6
7      // transition between coarse and fine grid level
8      scale(...);
9
10     // calculate advection diffusion of sediment
11     collisionSediment(...);
12     propagateSediment(...);
13     setFsSedi(...);
14     calculateSediConcentration();
15
16     // send sediment concentration
17     csWriter->send(cs);                    sediment concentration ⟹
18 }
```

Listing 4.11: Advection diffusion solver execution model, shown in pseudo code

Each of these C++ sub-solvers has been wrapped in a dedicated MUSCLE software agent. For the used setup, all sub-solvers use the same domain description for boundary conditions and lattice discretisation. Though different cell sizes are present in the non-uniform lattice, there is no scale separation between the three solvers (Figure 2.6). This also applies for the temporal scale, albeit the sedimentation process is slower compared

to the fluid dynamics. Here it is compensated by the internal model parameters which result in a faster motion (i. e. time-lapse) for the sedimentation process.

The coupling relations of the three sub-solvers are depicted in Figure 4.18. Herein, the sub-solvers have a circular dependency (section 4.1.2). The initialisation of the simulation has been kept together with the sedimentation solver, as it is done in the monolithic version. It starts with no sediment concentration within the fluid. Thus, there is no need to inject the initial data into the dependency circle from a separate agent.

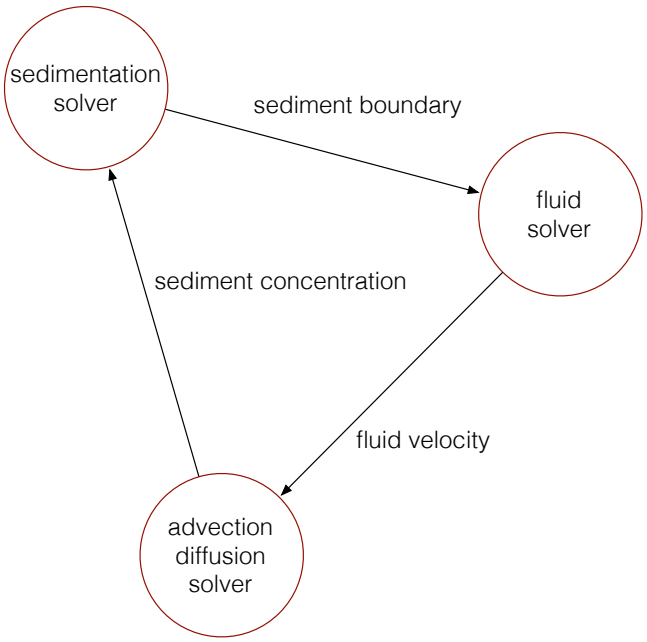


Figure 4.18: Coupling relations of the three sub-solvers of the sediment transport simulation

The calculation results of the coupled simulation implementation could successfully be validated against the original monolithic solver.

Interestingly, simulations with the coupled solver do finish a lot faster (i. e. within a shorter wall clock time) than the same simulations performed with the monolithic solver. This is mainly related to the time required for writing the result data to the hard disk. Because the coupled sub-solvers operate in parallel, dumping files to the disk becomes an asynchronous operation, during which the other solvers can proceed to compute their current time step.

As a successful coupled simulation can always be validated against the original monolithic solver, this coupling setup is part of the regression tests [130]. As such, it has been executed after each major implementation change of the MUSCLE framework. To facilitate an automatic red/green test, the results from the original monolithic solver are compared

by calculating the numerical difference for all lattice cells of the result data. Here the numerical difference utility *ndiff* has been used [10].

An early version of this coupling scenario has been published in [71]. Therein, all sediment related steps are represented by a single solver which is coupled to the fluid model.

4.4.2 In-stent Restenosis

Within the EU project COAST, simulating the biomedical problem of in-stent restenosis is part of the projects tasks. Herein, multiple numerical models have been coupled using the MUSCLE framework, each model dedicated to a specific aspect of the in-stent restenosis process. The sub-models will operate on completely different temporal scales in a range from seconds to days. Their spatial scales are ranging from $\mathcal{O}(\mu\text{m})$ to $\mathcal{O}(\text{mm})$. The coupled model considers a simplified approach of the complex process of in-stent restenosis and is explained in detail in [22].

Due to atherosclerosis, plaque can reduce the diameter of a blood vessel lumen. This narrowing is called stenosis. As a treatment, a balloon can be used to widen the vessel again. To further support the now injured vessel, a metal scaffold (stent) is additionally deployed to hold the vessel open.

In response to this treatment, the smooth muscle cells of the cellular tissue may grow abnormally. This causes a narrowing of the available vessel lumen, which produces a new stenosis. This additional stenosis is then called *restenosis* [82, 114]. Figure 4.19 shows such a restenosis: The intended widening of the vessel wall due to the deployed stent is reduced to a much smaller available vascular lumen because of regrown tissue cells (neointima). The stent may be coated with active compounds, which then elute from

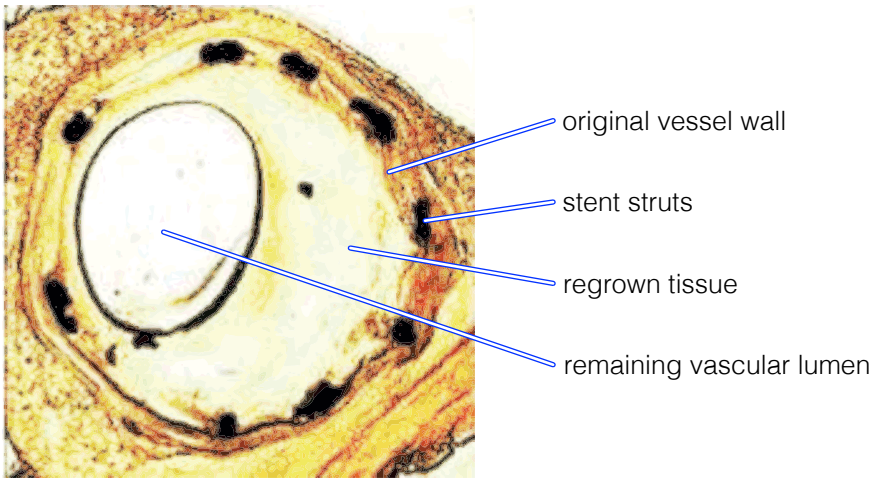


Figure 4.19: Stented coronary artery with a significant restenosis

the stent and are carried away with the blood motion. In contact with this drug, the

abnormal cell grows around the stent may be reduced.

The thickness and shape of the stent, as well as the influence of the drug are key factors for a successful treatment of atherosclerosis. This process should thus be modelled and studied with a numerical simulation, to be able to eventually evaluate the interaction of these properties.

One of the main parts of this multi model and multi scale scenario is the smooth muscle cell behaviour, which simulates the cell growth, interaction with the drug and interaction with other cells. Another part is the blood flow, which has an direct effect on the drug concentration and applies forces to the cells of the tissue. The diffusion of the drug is another separate solver in this application [22, 49].

Cell Proliferation

In the cell model, the cells begin to proliferate in response to external mechanical forces. These are determined by the wall shear stress and oscillatory stress index, which are results from the blood moving along the cells. The oscillatory stress index is a measure of the degree of the oscillating movement of the blood in a vessel due to cardiac cycles.

The cell proliferation solver has been written in C++, implemented as an multi agent based system (Table 4.3). Each cell is represented by an agent rule set, which processes the individual life time cycle for each cell. This life cycle dictates the temporal scale of this model, which is in the order of days (Figure 4.20). This solver has been implemented within the scope of the EU project COAST [22].

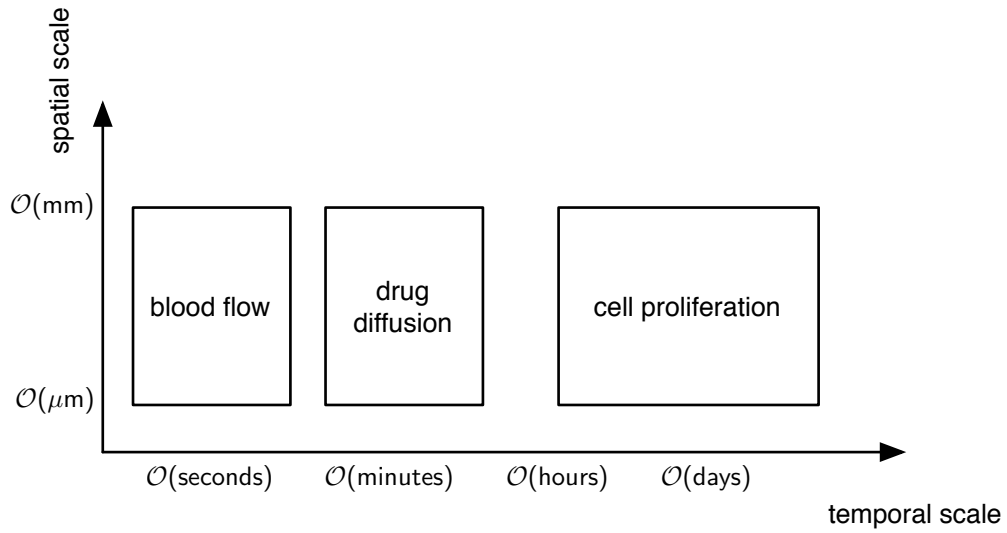


Figure 4.20: Scale separation map of the in-stent restenosis model

New cell agents are created when mitosis (cell division) occurs, which is determined individually for each existing cell. The proliferation is driven by low wall shear stress

or a high oscillatory stress index, but may be stopped depending on the current drug concentration and contact inhibition of adjacent cells.

The cell proliferation model knows the cell positions and sizes, as well as the number of neighbour cells. The required inlet information is the wall shear stress and oscillatory stress index, which are being sent by the blood flow solver.

The growing cell domain has direct influence on the blood flow domain, which has to shrink accordingly. Similar, the diffusion of the drug within the cell area is confined to the domain of the cell domain. Hence, the cell proliferation model passes the current boundary description to a MUSCLE outlet.

Blood Flow

The blood flow is modelled as an incompressible Newtonian fluid, whose numerical solution is calculated using a Lattice Boltzmann based solver. This solver has been created within the *International Lattice Boltzmann Software Development Consortium*. The implementation uses a highly optimised Fortran code, tuned to run with a maximum performance on NEC-SX vector machines. As such, this sub-model is a parallel solver in itself. It can also be used on distributed memory cluster systems, where the internal communication is based on MPI [8].

Regarding the temporal scale, this is the fastest of the models. Its temporal scale is determined by the length of a cardiac cycle, i. e. $\mathcal{O}$(seconds).

The solver uses a uniform grid discretisation, whose boundary depends on the growing cell area. The more the cells proliferate, the smaller gets the size of the remaining vascular lumen for the blood flow. Information about the transient interface to the cell model is received via an inlet during computation.

The output for the blood flow solver consists of the wall shear stress and oscillatory stress index.

Drug Diffusion

After the stent has been deployed into the vessel, the drug is eluted from the stent struts and diffuses into the surrounding tissue. In contact with the cells, the drug can inhibit cell proliferation. To determine the drug concentration within the cell area, this model solves an anisotropic diffusion equation using a finite difference approach [22]. Herein the stent struts act as a source for the drug, which is carried away by the blood flow (sink). Thus, the drug diffusion model shares the computational domain with the cell proliferation model.

Required input for the drug diffusion model is the current geometry of the tissue domain, which is obtained from the cell proliferation model. As output, the model sends the drug concentration for a set of discrete positions.

To reach a steady drug concentration within the tissue, the time scale for the drug diffusion process is in the order of several minutes [110]. Together with the dimensions of arterial

tissue and the diffusion coefficients, this leads to a temporal scale of the drug diffusion model in the order of a few minutes. Hence the coupled model uses the steady drug concentration ($\mathcal{O}(minutes)$) as input for the much slower proceeding cell proliferation model ($\mathcal{O}(days)$).

The drug diffusion solver has been explicitly created for the purpose of the ISR simulation and has been implemented using the Java programming language.

sub-model	blood flow	drug diffusion	cell proliferation
language	Fortran	Java	C++
hardware	optimised for NEC-SX, with SUPER UX	single CPU	single CPU
spatial scale	$\mathcal{O}(\mu\text{m})$ – $\mathcal{O}(mm)$	$\mathcal{O}(\mu\text{m})$ – $\mathcal{O}(mm)$	$\mathcal{O}(\mu\text{m})$ – $\mathcal{O}(mm)$
temporal scale	$\mathcal{O}$(s)	$\mathcal{O}$(m)	$\mathcal{O}$(h) – $\mathcal{O}$(d)
domain description	Eulerian	Eulerian	Lagrangian

Table 4.3: In-stent restenosis sub-solver characteristics

Stent Deployment

At simulation startup, the stent is deployed into the vessel: The stent is unfolded and pushed into the vessel walls, which causes ruptures in the tissue.

The deployment of the stent is modelled in a separate module, which passes the motion of the stent struts to the cell proliferation model (Figure 4.21). Now the cell positions around the struts are determined by the cell proliferation model and the simulation cycle starts.

Data Mapping

In the first versions of this coupling setup the data has been mapped within the conduits. After some related test series, the conduits were replaced by separate minimal sub-solver agents. As the conduits, they are only used for the data mapping, but as any sub-solver, these mapper agents are allowed to own multiple inlets and outlets. The reason for this was to reduce the computational cost for the mapping procedure and also reduce the amount of data being passed between the sub-solvers. This way, the mapping between the different domain representations of the three solvers could be achieved more efficiently. For example, for the mapping of drug concentration, the mapper agent receives the lattice based (i. e. Eulerian description) drug concentration from the drug diffusion simulation and also the cell locations from the cell proliferation solver, which uses a Lagrangian model. Now, the mapper can calculate a accumulate a single drug concentration value for each cell.

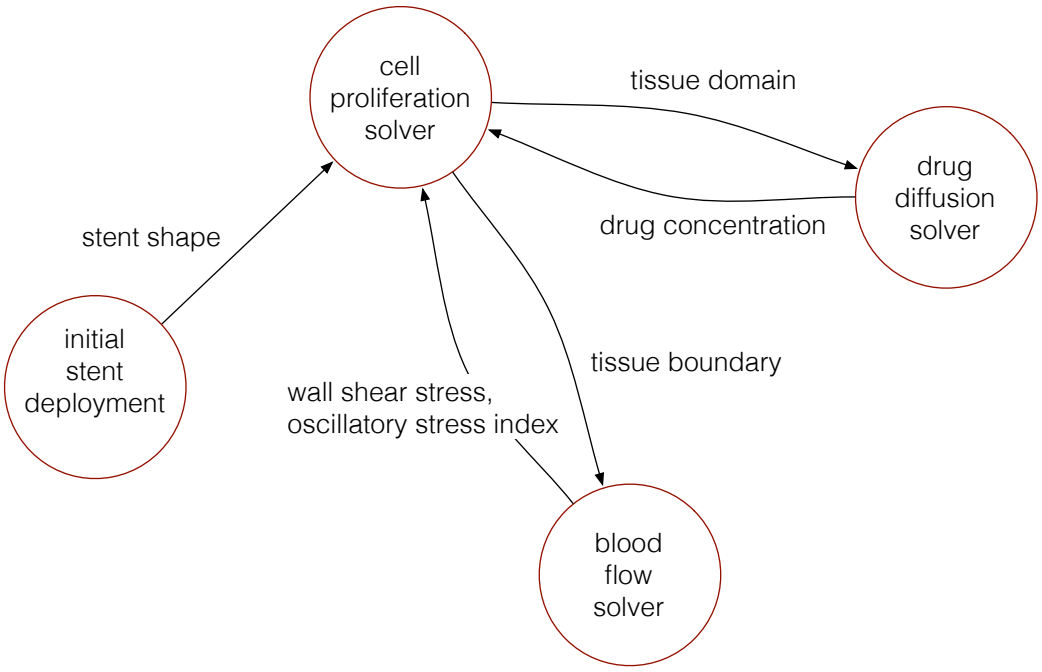


Figure 4.21: Coupling relations of the in-stent restenosis simulation

As the cell proliferation model was newly written for the in-stent restenosis simulation, these mappings could also have been implemented within the cell proliferation model. But according to the idea of MUSCLE, each sub-solver should not require algorithms for a specific coupling setup.

Simulation Results

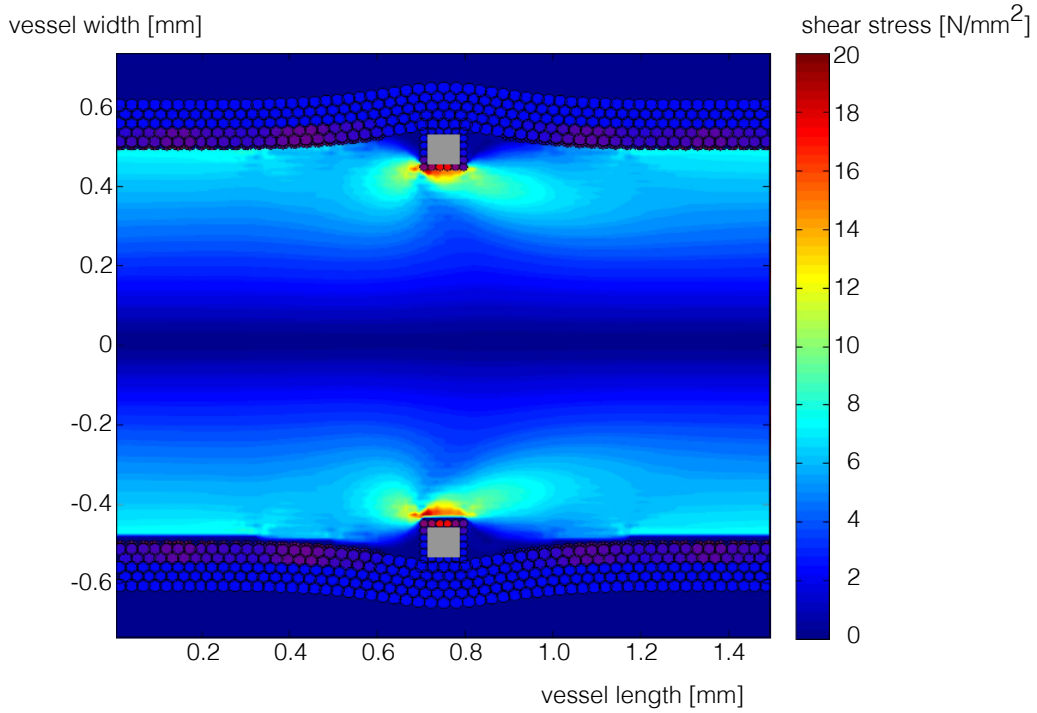
The coupled simulation has been computed for several settings. The primary configuration domain involved a vessel section around a strut segment. The considered vessel has a length of 1.5 mm and a width of 1.24 mm (Figure 4.22). The wall of the vessel has a thickness of 120 μm. The cells within the tissue area start with an average radius of 15 μm and are densely packed at the vessel wall.

Two square struts of side length 90 μm are being deployed at initialisation. They are pushed through the confining layer of cells, which is called internal elastic lamina [22].

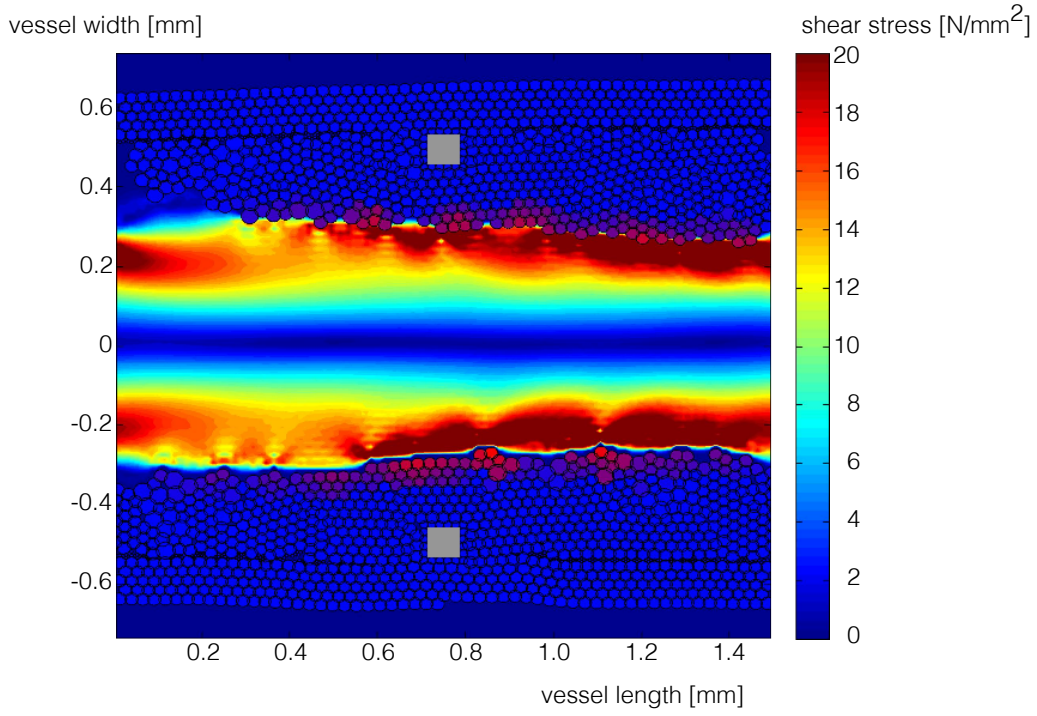
Two machines have been used for the computation:

- an Intel Woodcrest machine with 8 physical cores on 2 dies
- a NEC SX-8 vector computer with 4 CPUs

The computation intensive blood simulation has been performed on the SX.8, whereas the other parts of the simulation were run on the Intel machine.



(a) Stent has been deployed and cell configuration determined, blood flow and drug concentration at steady state



(b) Condition 28 days after stent has been deployed

Figure 4.22: In-stent restenosis simulation for drug eluting stent

These preliminary results show a proliferation of the tissue in response to the injury resulting from stent deployment. The grow area is pushed behind the main injury zone at the stent due to the blood flow. The remaining vessel lumen is clearly reduced compared to the start of the simulation and causes an increased shear stress. An equilibrium is reached because the cell proliferation is inhibited if the shear stress reaches a certain threshold.

The simulation has also been performed with a bare metal stent (i. e. no drug model), which results in a even more narrowed vessel lumen because of an about 7 % thicker tissue layer [22].
In the present simulation, the model parameters are based on porcine data, for which similar results could be observed *in vivo* [149].

Given the design of the MUSCLE framework, it has been inherently simple to couple the different sub-model implementations and to perform sensitivity studies using the inheritance mechanism of the configuration files (section 4.2).

4.4.3 Massively Parallel LBM

MUSCLE has also been explored in a large scale computation environment which was subject to the thesis research of S. Freudiger [58]. Herein, a distributed memory parallel flow solver has been created which uses the Lattice Boltzmann method to solve the Navier-Stokes equations numerically (section 3.3.2.1). The software groups individual vertices to so-called *blocks*. At runtime, these blocks are being used to perform optimised bulk communication of the particle distributions to adjacent blocks. Internally the framework uses MPI to perform this communication of mass data, but it can also operate without MPI. In this case, it uses the RCF communication library (page 3.3 and [112, 113]).

Benchmarks have been performed to test how the overall simulation performance scales with different optimisation strategies of the blocks. In addition to the MPI and RCF communication layers, MUSCLE has also been compared against these speed optimised message passing libraries.

Consequently, the sub-solver tasks running on individual machines have been wrapped in a MUSCLE agent. For the Lattice Boltzmann solver, a message passing hook has been developed, which allows to forward all the communication via inlets and outlets of the respective MUSCLE agents.

The benchmark compares three different aspects:

- How well the system scales with an increasing number of machines and at the same time an increasing computational domain. This technique of *weak scaling* is a standard procedure to measure the parallel *scaleup* (section 2.2.4). A domain of 100^3 lattice points has been used per machine, with four different setups: 1, 8, 12 and 16 machines respectively.
- Different optimisation strategies of the block communication have been compared

(i. e. pooled and non-pooled communication [58]). But also different number of blocks per machine have been compared: One single block per machine (i. e. 100^3 vertices) has been compared to a situation with many block per machine. This second case involved a number of 10^3 blocks per machine, which results in 10^3 vertices per block.

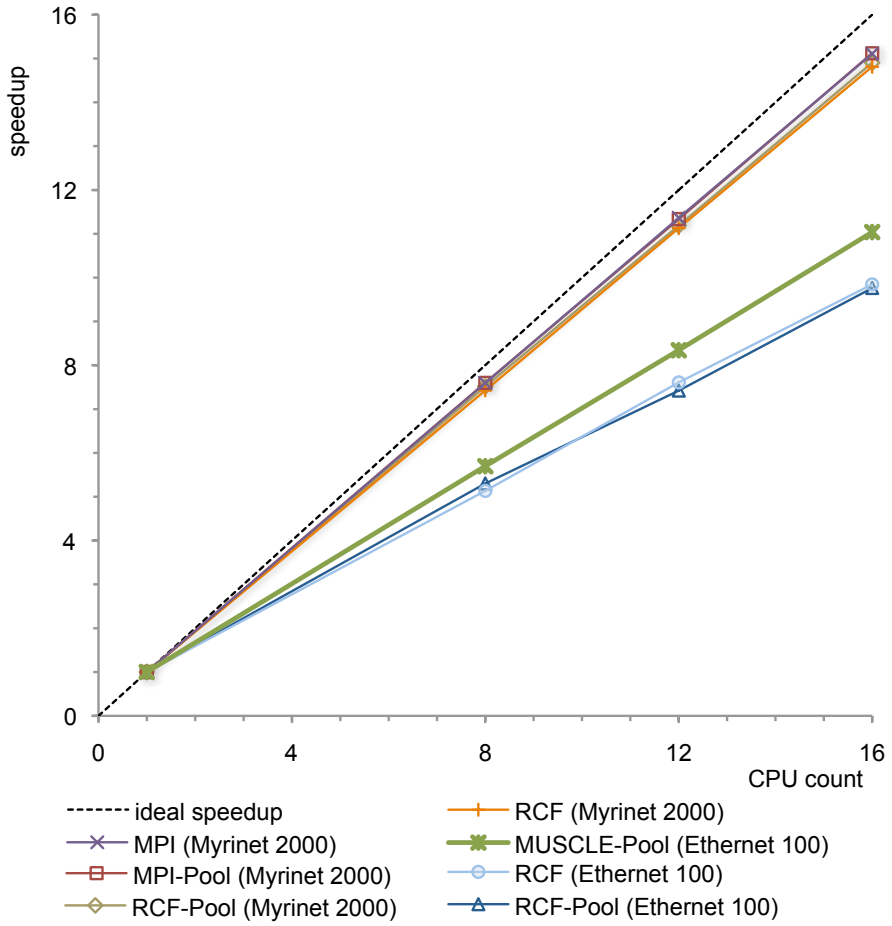
- All the variations have been performed using MPI, RCF and MUSCLE as the communication framework.

The computation has been performed on a PC cluster system consisting of 60 interconnected machines [5]. Each machine features two CPU cores of 64 bit AMD Opteron, 1.4 GHz. The machines are connected via an Ethernet network (100 Mbit/s) and a high speed Myrinet2000 connection (2 Gbit/s) [131]. The operating system is a 64 bit Debian 3.1 [44] and 64 bit Ubuntu server edition 8.04 [23].

Figure 4.23 shows results for the computations performed with one large block per machine. It has to be noted, that due to technical issues on the particular cluster system, the MUSCLE simulations could not be performed using the Myrinet 2000 network interface.

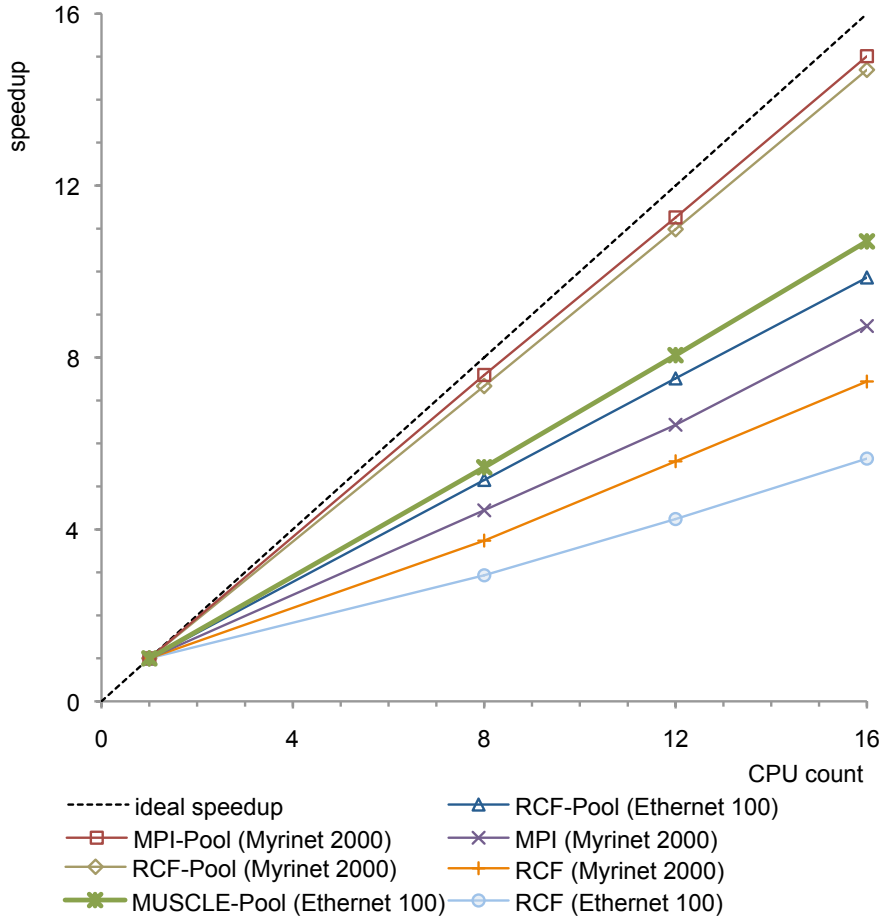
When these benchmarks have been performed, the message passing mechanism of MUSCLE did not yet allow to omit the conduits for a connection in case that no mapping is required. As the conduits itself are controlled by JADE agents, this means an unnecessary high number of agents in the system and also a detour for the messages. For each of the discrete particle directions a block has 18 adjacent blocks to be considered (Figure 3.2). This leads to 18 outlets and 18 inlets per agent, each connected to a conduit. Despite of that, MUSCLE performs quite well for this benchmark and it scales linear with an increasing number of agents.

The second setting uses the configuration with 10^3 blocks per machine as described above. This poses an interesting challenge for MUSCLE, as now the number of outlets, inlets and thus the number of conduits strongly grows. Hence, there have been $2880 \cdot 2$ inlets and outlets per MUSCLE agent [58]. Here, MUSCLE scaleup remains similar to the one of the previous setting, whereas some of the other configurations have shown a worse performance (Figure 4.24).



modified from [58], Figure 17.22 a

Figure 4.23: Virtual Fluids, 100^3 lattice vertices per sub-solver



modified from [58], Figure 17.22 b

Figure 4.24: Virtual Fluids, $10^3 \cdot 10^3$ lattice vertices per sub-solver

5 Coupling With Automatic and Mutable Connections

After the positive experiences with MUSCLE with respect to flexible coupling setups in HPC environments, the possibility of an even more flexible, approach has been explored.

In MUSCLE, the coupling interactions are defined a priori within the configuration file. This is perfectly reasonable if the number of coupling interactions is limited, e. g. $\mathcal{O}(10)$, or if they can be generated automatically with a partitioning library (section 2.3.1). Thus, a mechanism has been developed, which can couple an arbitrary number of sub-models without the need for an explicitly, a priori defined coupling graph. Herein, the coupling dependencies are determined at runtime using a pattern matcher strategy. The coupling mechanism has been extended to enable sub-simulations to be added or deleted in the running simulation.

This framework could bring the agent based HPC to areas such as environmental simulations, where a configuration of sub-models may depend on the results of subsequent computations of other sub-models. For example, a model simulating the wave height and water level at oceans may only spill water onto coastal buildings, if a dike or similar structure becomes over-flooded. In such a scenario, it is not clear beforehand which models participate on the solution and how they are coupled. Instead of the a priori, top-down approach of MUSCLE, this would require a bottom-up solution with self organising coupling interactions.

First, it has been tried to modify MUSCLE to adapt to this new concept. But soon it became apparent, that the JADE library used within MUSCLE imposes some limitations. The agent control instance of JADE, the so-called *agent monitoring service* (AMS) sometimes produced deadlock situations, preventing to add new agents to a running simulation. Adding agents at any time during the simulation is one of the features which should be explored within this further coupling library development, and thus JADE does not seem to be suitable anymore.

For a general multi agent based approach, one of the big advantages of JADE is its FIPA compliance. But this becomes a bottleneck when targeting very large scale distributed HPC applications. Within the FIPA specification, the agent messaging strives to be as compatible to arbitrary other agents as possible. Because of this, the message of each communication is expressed using a dedicated content language, the agent communication language (ACL) [1]. These messages have a very verbose preamble attached, which is not required if the messages are mainly used to communicate data messages within the same

framework.

After some discussions with the JADE developers, it became apparent that JADE is probably not a suitable basis for the extended coupling approach. JADE is very tightly woven to the FIPA concepts and removing them from the JADE implementation poses to be more complex than writing a specialized library anew. Therefore, a new agent framework has been designed and implemented to provide a solution. Its internal work name is Bond and technology wise, it can be considered the successor of MUSCLE regarding HPC computing. It allows dynamic coupling setup which may change at runtime. New agents may be introduced to a running simulation at any time. Also an a priori statically defined connection scheme is not required. The connections between individual agents will be created at runtime.

5.1 The Bond Message Passing Strategy

In Bond, the message passing strategy has been arranged in a manner which consequently delivers messages asynchronously. This results in a fast throughput, even if sub-simulations are added during the computation (Figure 5.3). Apart from the standard remote message passing (left branch in Figure 5.3, remote send with unguarded data), Bond can efficiently pass data to agents which are being executed on the same machine. This is important to effectively utilise multi core machines (Figure 5.9).

Sending Bulk Data

The Bond message passing back-end has been designed and implemented with HPC applications as the main target. Typical HPC applications have to deal with two major resource constraints: duration of the calculation (speed) and the amount of available memory (RAM). Both are the main reasons to use distributed software for calculation in the first place. So especially in the areas of speed and RAM, Bond should not add any considerable overhead to the final distributed application. As future hardware will allow to run applications which will consume more and more RAM and CPU time, a constant coupling overhead will become more and more negligible.

The critical sections of the application runtime, where resource management affects speed or RAM, are the send/receive calls. These often happen frequently over the lifetime of the program.

A minimal send operation consists of the attributes *what* and *where*. The *what* is an arbitrary data object (e. g. an array of double values) and the *where* is the address of the receiving side. To perform a send call, Bond has to figure out how to contact the receiver and then deliver the data (Figure 5.1). Figuring out the route to the target is a constant operation which will only be run once. Memory usage for this operation is minimal and varies slightly with the number of agents in the platform. For successive send calls, the cached route from the first call will be used which produces no further delay. In the case of a local send a reference to the data is passed, which is a constant operation. For

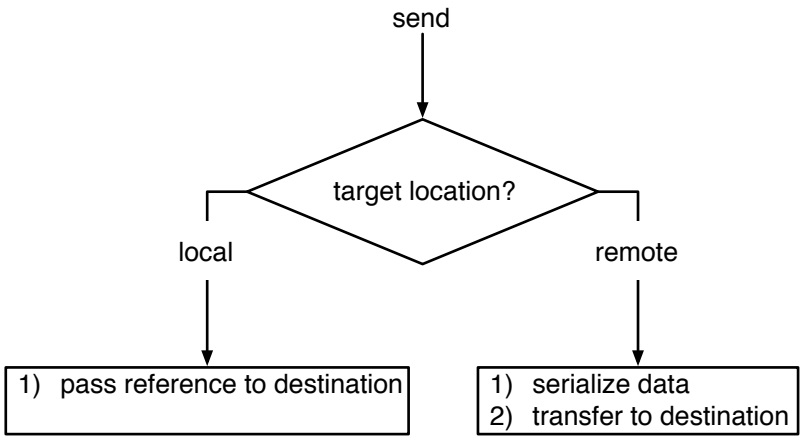


Figure 5.1: General steps of a send procedure

remote sends, the data has to be serialised/marshalled (linear operation), transferred over a network (linear operation) and unmarshalled on the receiving side (linear operation). Here the sending side requires only a constant amount of memory, because the data can be streamed to the network device (size of the stream buffer). Similarly, the receiving side requires a constant amount of memory as well.

From the Bond point of view, a *send* command is just another specialised command which will be processed by the asynchronous command delivery system (5.1). Thus calling the *send* method will return immediately letting the model code continue at once.

Communication within the Bond Platform

From the user point of view, the Bond platform provides one to many software agents. The agent wraps a numerical solver and provides a programming API to communicate with other Bond agents. The agent programmer does not have to care whether these agents are running on the same or a remote machine, as the usage of the API remains the same. At launch time it can be decided how to distribute the required agents on the available hardware. Each agent can open one or more channels to send and receive data from/to a dedicated peer. Because the communication speed between the agents should not depend on the number of agents, each agent to agent communication (i. e. send/receive) pair uses a dedicated unidirectional connection whose route is only configured once. Combined with the peer to peer (P2P) connected outposts this results in a well scalable setup. Figure 5.2 shows the key components of the communication mechanism. The agents are attached to an outpost, which holds an IP address at a random free port to allow remote communications. The various outposts (at least one per participating machine) announce themselves to the headquarter, which is accessible via a given port on the responsible machine. A single headquarter administrates all outposts of a Bond platform. The headquarter maintains a white page service responsible for consistent agent naming and is also host for the router component.

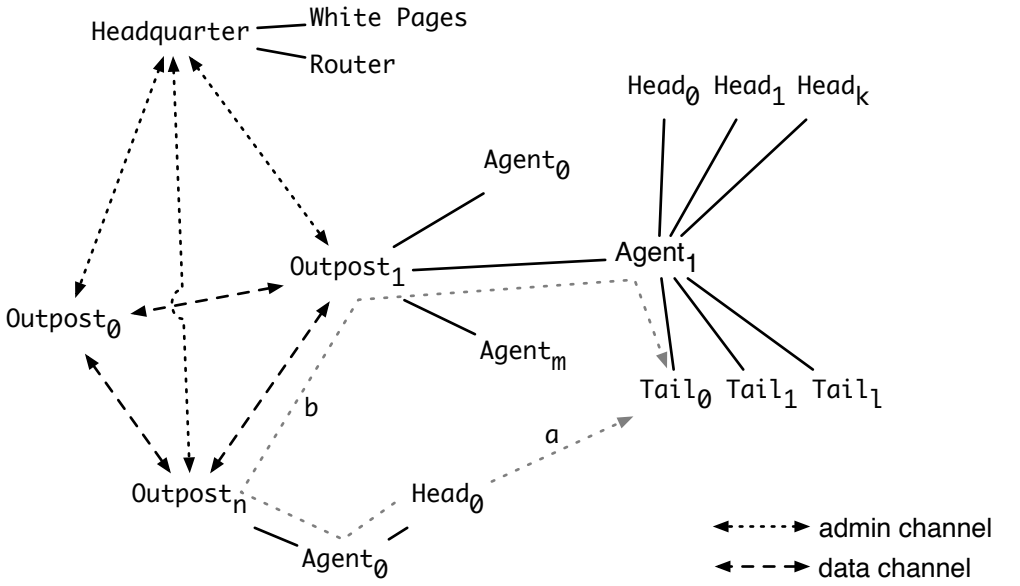


Figure 5.2: Communication components of a Bond platform

The router is an optimised yellow pages service: it collects descriptions for head and tail components and determines the matching pairs. The head and tail components are the endpoints for a data transfer. Heads are on the sending side, whereas tails are responsible for receiving the data. The heads and tails are the handles accessible to the user to issue a send/receive call. Calling the send method on a head and the matching receive method on its connected tail is the only thing to be done in the solver code to transfer data (line a) in Figure 5.2). Internally, the Bond framework adds the send command to the command queue of the outpost of the agent.

Each outpost uses a command queue to handle all platform communication asynchronously, like it is described by the command design pattern [60, 20, 148]. The queue is a FIFO queue with multi thread support. This allows to increase the number of threads which process the queue if this would lead to faster throughput on the given hardware. Thus there has to be only one network connection for each pair of outposts, rather than a connection for each pair of agents. For n outposts this would lead to a number of

$$\frac{n(n-1)}{2} \tag{5.1}$$

network connections, making the logical network topology a fully connected mesh (also known as *complete graph*). From a performance point of view, this is also the ideal topology for a physical network. On the receiving side, a thread pool is being used to answer each incoming remote command instantly. Using threads at this point, the data can be passed to its designated tail in a non-blocking manner. This is crucial to avoid a deadlock situation if the order of send and matching receive calls is not nested. This

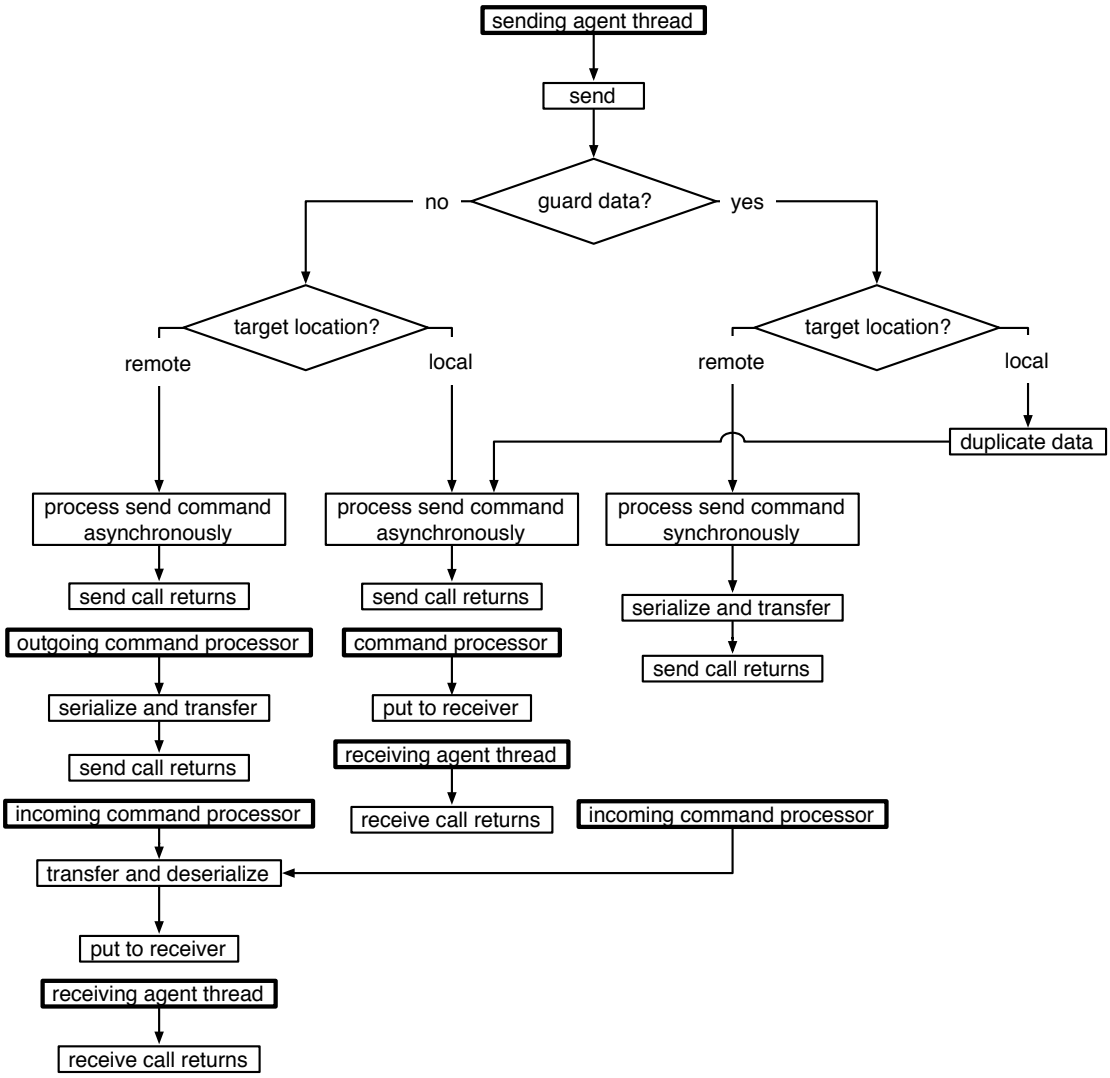


Figure 5.3: Asynchronous send/receive procedure in Bond

even allows an agent to send data to itself, without blocking its own thread of execution. Because the threads are managed using the pool pattern [138, 148], their creation and management requires only a minimum of CPU resources.

5.2 Benchmarks

As Bond has not been used in as many simulation scenarios as MUSCLE, a set of benchmarks has been performed to estimate its capabilities. These are artificial scenarios, whose coupling interactions can be found in realistic simulations, but therein the measured duration (wall clock time) of the programme will mostly depend on the speed of the sub-models.

5.2.1 Branching Rivers

In this section the ability of Bond to dynamically connect agents as they are added to the whole simulation is being evaluated. The setup is created by subsequently instantiating sub-simulations, which represent simplified river models. Each river is a section of a bigger branched stream and has an incoming flow rate and a outgoing flow rate. This setup has been chosen, because it can be compared with a different number of involved sub-models.

In the simulation, the outlet description of each river contains the geographic coordinates of the river end-point. Now the simulation is being started, and river agents are added. Each river owns a single outlet, which is being announced to the Bond framework. Additionally, each river announces a dynamic inlet to Bond. This contains a description of the whole river, from start- to end-point and also an algorithm to determine which external outlets can be connected. These would be the end-points of other rivers. The dynamic inlets can spawn multiple inlets on demand to accept connections from other rivers.

If a matching river agent is available, i. e. a river which has its outlet on the path of another river, a connection is being established (Figure 5.4).

The individual rivers have been created in a way that all rivers (except a single one) end on another river. This tests two abilities of Bond:

- the ability to *automatically connect* outlets and inlets at runtime
- the ability to *dynamically create* inlets at runtime

The tests have been done with settings up to 850 rivers (Figure 5.5). During the tests, the river agents have been executed in parallel on a cluster computer. Figure 5.6 shows the chronological order in which the river outlets have been connected to the receiving rivers. This depends mainly on the time when the corresponding agents have been launched into the simulation, but also of the speed with which the Bond framework processes the outlet and inlet announcements.

These announcement requests are processed in the *router*, a module of the bond man-

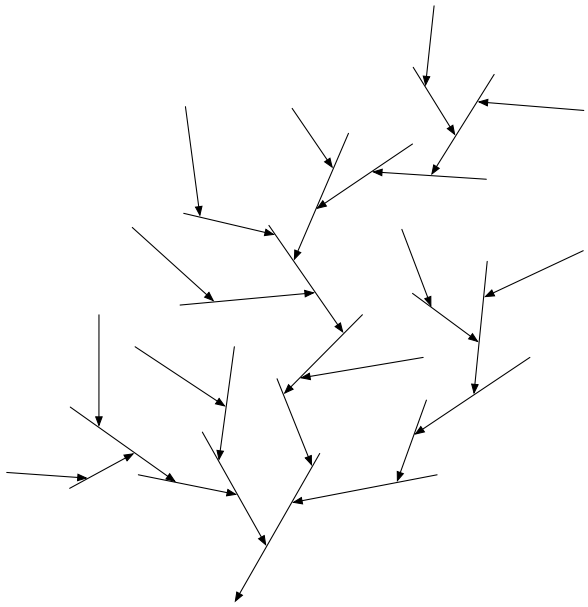


Figure 5.4: Automatically connected branch of 32 rivers

agement which itself has been implemented with a shared memory parallel approach. As there have been performance problems with JADE when many agents are added dynamically during the simulation (page 99), the speed of the Bond routing activity has been measured (Figure 5.7). Bond can process the connections of $\mathcal{O}(100)$ agents from different machines quite well, due to its internal parallel routing management. The actual duration to establish outlet/inlet connections also depends on the implementation of the matching algorithm within the dynamic inlets, which computes in constant time for the river implementations. If the rivers are launched in random order, the router has to apply the matching algorithm for all outlets on all other available rivers, i. e. $\mathcal{O}(n^2)$.

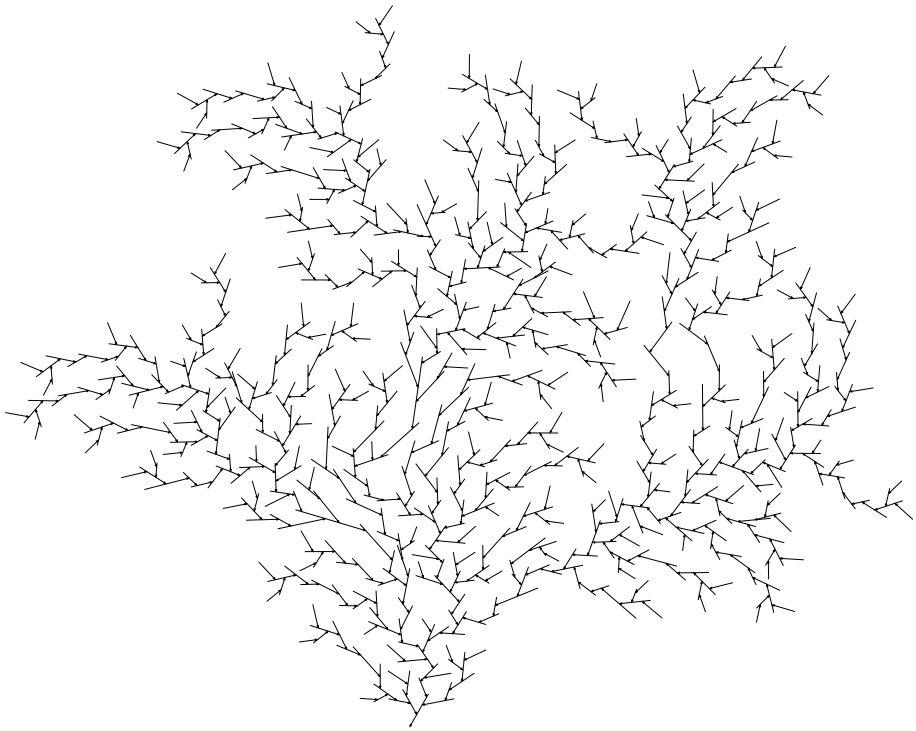


Figure 5.5: Automatically connected branch of 850 rivers

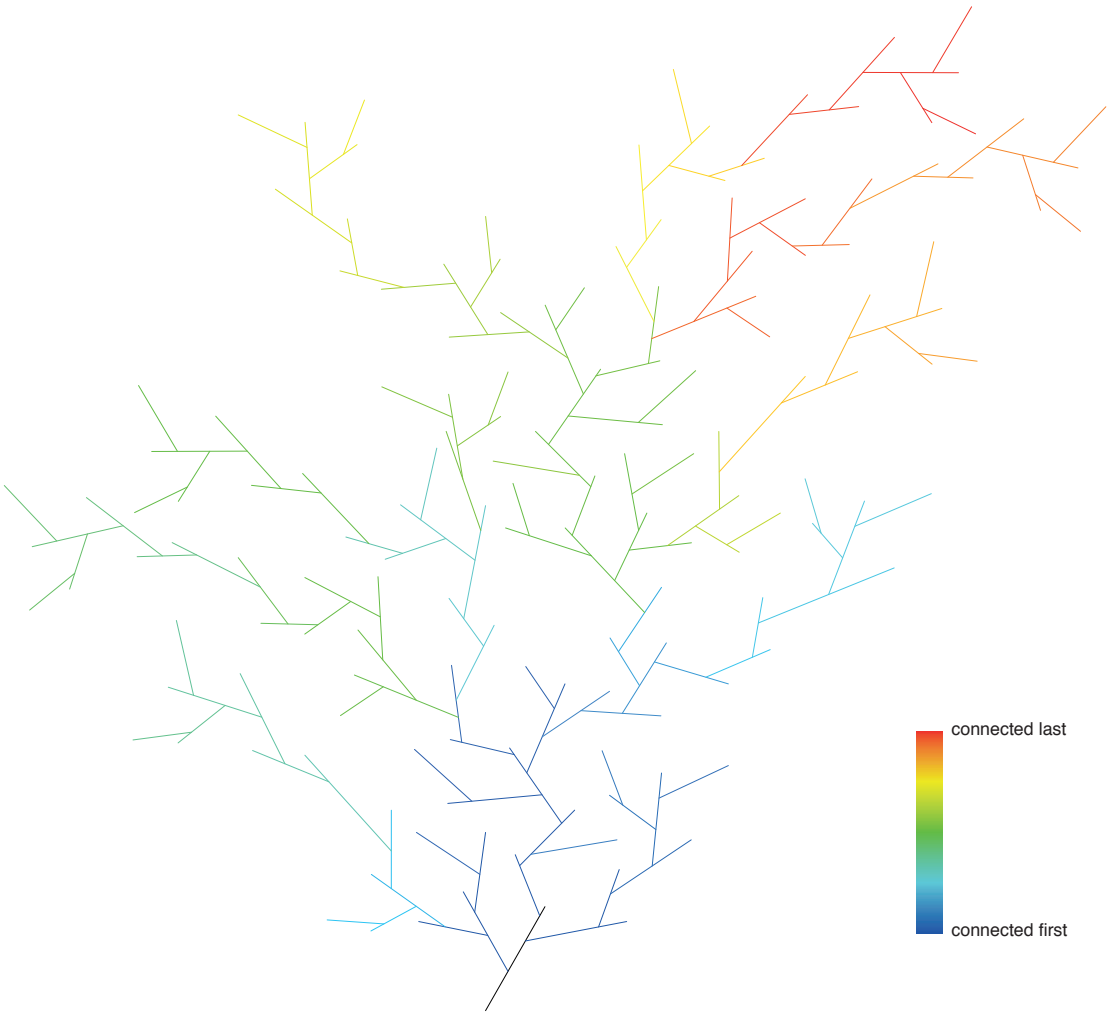


Figure 5.6: Rivers coloured chronologically with respect to their outlet connection times (150 agents)

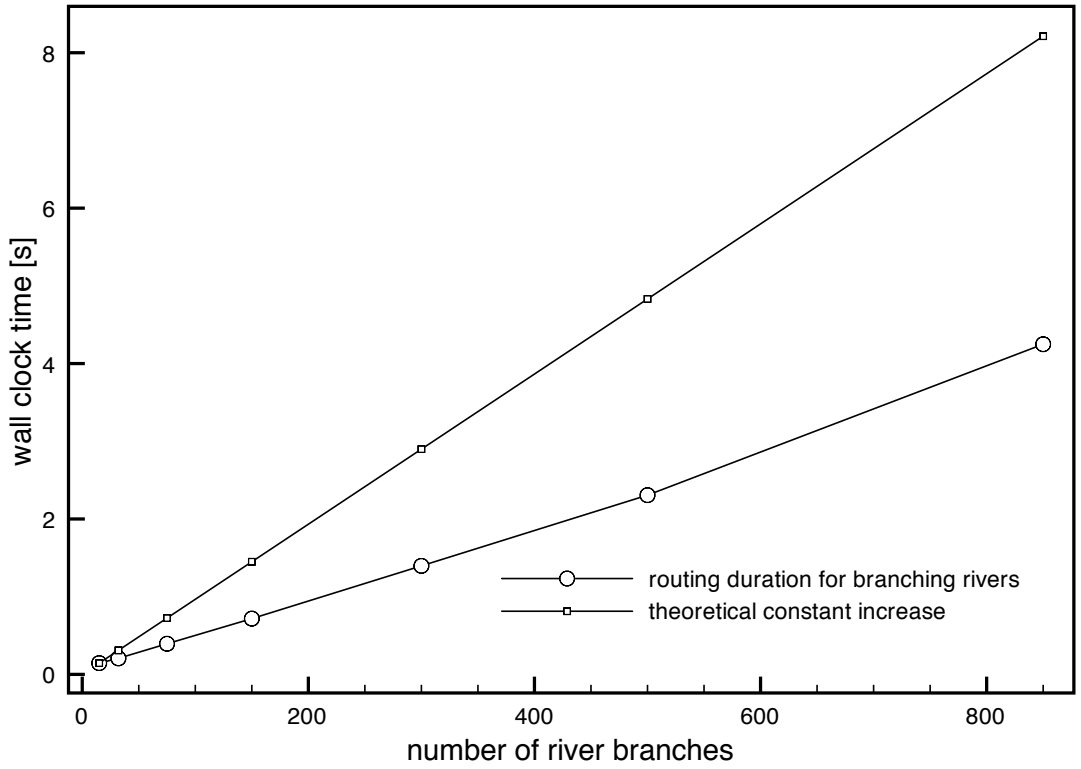


Figure 5.7: Duration of routing activity to dynamically connect 15, 75, 150, 300, 500 and 850 branched rivers. As a comparison, the theoretical duration for a reference time of a respective multiple of the first value (15 branches, 0.145 s) is shown, i. e. $\frac{850}{15} \cdot 0.145\,s = 4.249\,s$.

5.2.2 Ringtest

One important aspect of a coupling framework is its raw message passing speed, as it has a direct influence on the overall parallel efficiency (section 2.2.4). Being a higher level architecture as e. g. the MPI library, it should nevertheless provide a throughput with the same order of magnitude. This ought to scale with an increasing number of peers, meaning the internal communication design does not contain any sequential bottlenecks.

For a rectangular/quadratic compute domain, a very simple partitioning algorithm could just split the domain in one direction into equally sized segments. This kind of partitioning scheme is actually being used in distributed scientific computations [102].

If the outer left and outer right edges of the global domain exchange data too, also known as periodic boundary, the communication topology of the partitioned domain will look like a linked ring. This setup is suitable to estimate the management overhead the communication library will impose on the distributed program. Any HPC grade communication library should

- use as little computation time as possible
- keep at least constant time overhead when number of peers scales from one to many
- use as little memory as possible
- keep at least constant memory overhead when number of peers scales from one to many

In this setup, one of the peers sends a data message to its neighbour peer in the ring. As soon as the data is available in the target peer, it will itself send a data message to the next neighbour. This will continue until the starting peer will receive a data message itself. This test has been implemented as

- a Java programme using Bond agents
- a Java programme using MUSCLE agents
- a C++ programme using the MPICH2 MPI library

The measured results are shown in Figure 5.8, depicting the wall clock times for 10, 20, 30 and 40 participating machines. These tests have been run on the homogeneous PC cluster described on page 95. It is observed, that the performance of MUSCLE could be much improved with the Bond framework, almost reaching that of an MPI library.

Today HPC clusters often offer multiple CPUs or cores and can execute multiple threads in parallel via simultaneous multithreading (e. g. Intel's hyper-threading). Therefore the setup has also been executed on a machine with 8 cores available (Figure 5.9). As MPICH2 uses the loopback device (a virtual TCP network interface) to "send" the data to the machine itself, the results depend on the operating system [76].

The most common scenario will probably be a hybrid setup, where multiple machines are connected, each offering multiple CPU cores. The results for a hybrid approach are shown

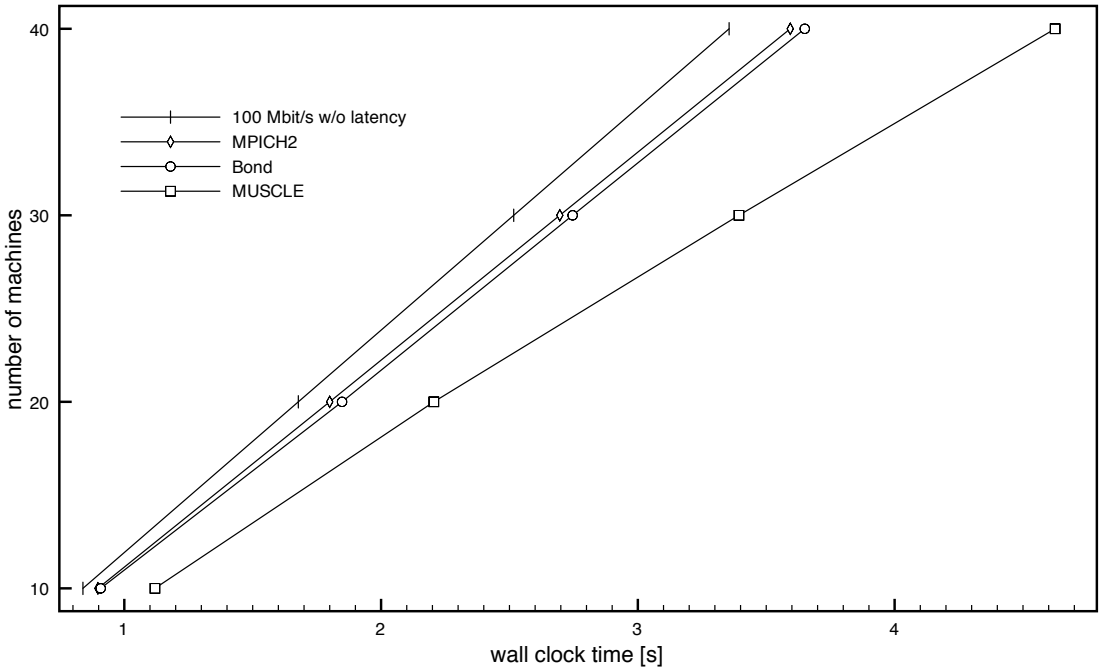


Figure 5.8: Ringtest with 10 to 40 machines using a 100BASE-TX network. Message size has been 1 megabyte. Mean time for 10^4 round-trips.

in Figure 5.10. Here each machine executes two tasks of the coupled programme. Here, the improved shared memory approach of Bond even leads to slightly better performance than the MPI library. This test has also been performed on a larger scale cluster, where up to 112 machines have been used, each using all of its 8 CPU cores, leading to 896 coupled agents (Figure 5.11). MUSCLE has not been used in this comparison, as it could already be shown that the Bond implementation yields faster results.

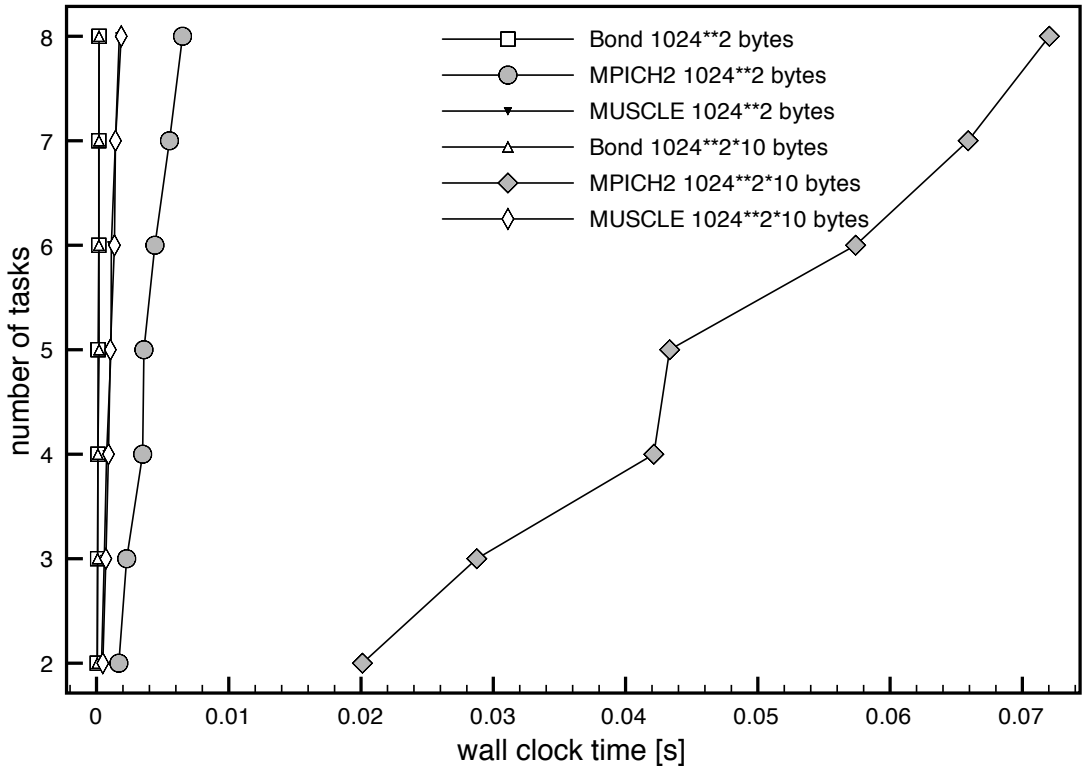


Figure 5.9: Ringtest with 2 to 8 parallel tasks on a single machine (8 CPU cores available). The message size has been 1 megabyte and 10 megabytes respectively: mean time for 10^4 round-trips.

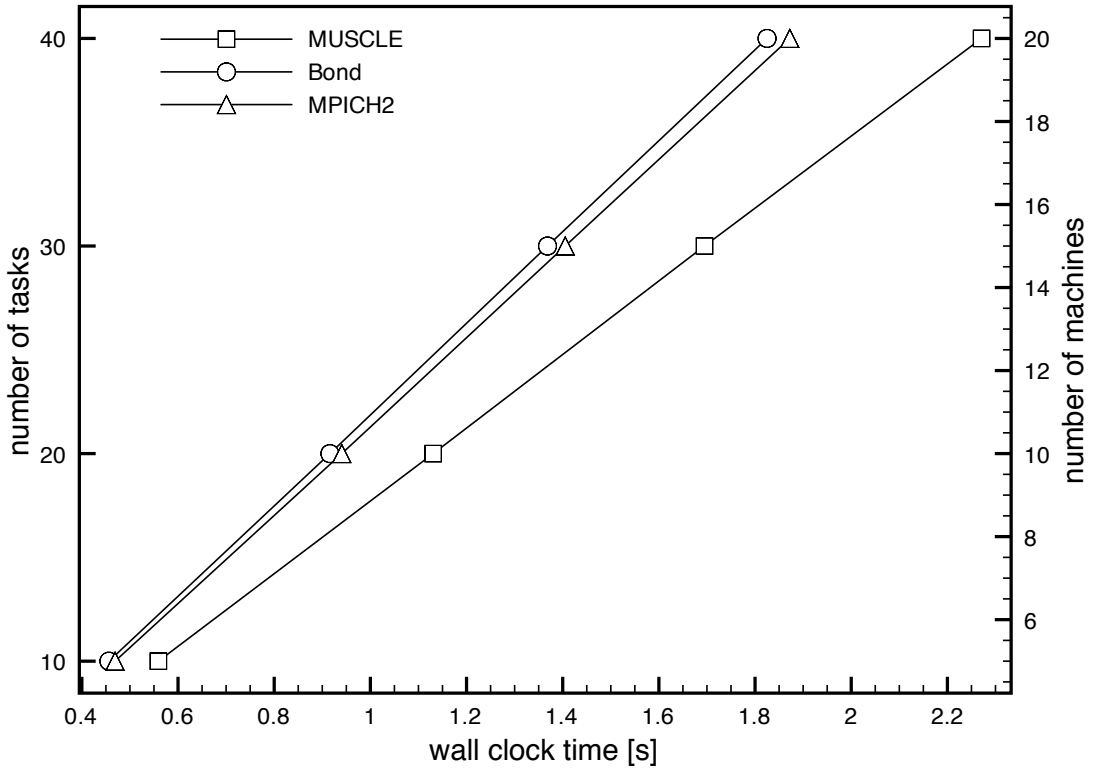


Figure 5.10: Ringtest with 2 tasks per machine (2 single core CPUs per machine available). The message size has been 1 megabyte: mean time for 10^4 round-trips.

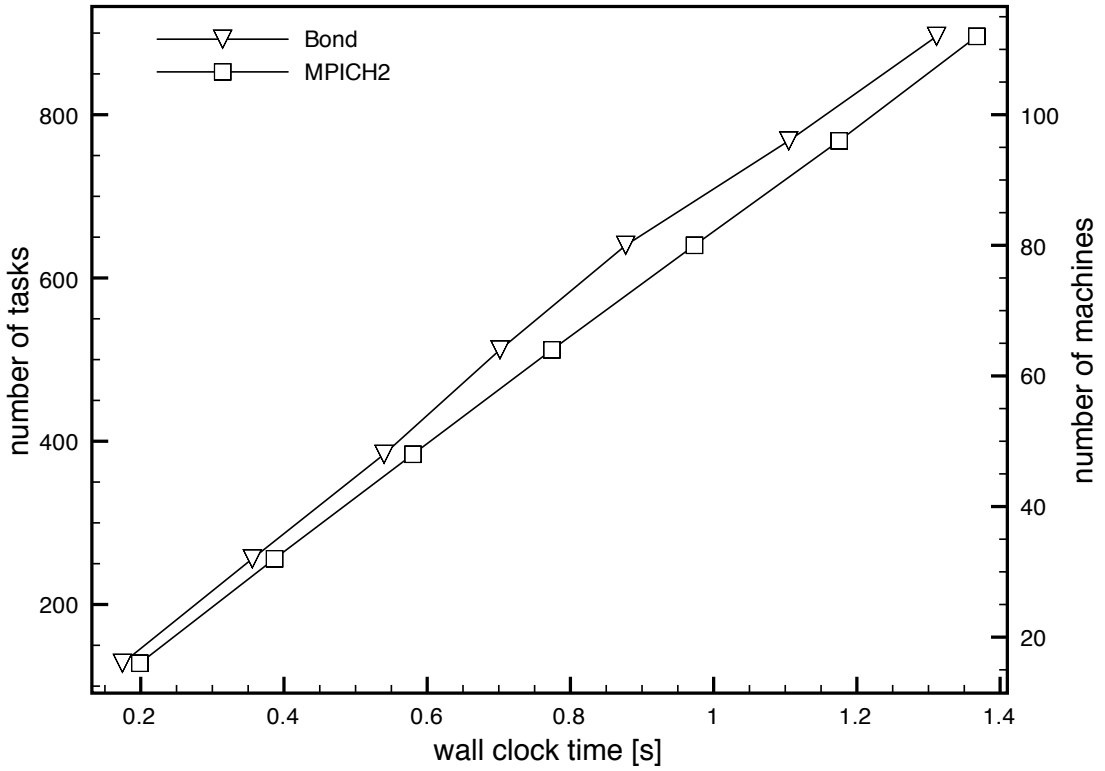


Figure 5.11: Ringtest with 8 tasks per machine. The message size has been 1 megabyte. Each machine has 8 CPU cores available: mean time for 10^4 round-trips.

6 Conclusions

The following section will give a brief summary of the work achieved within this thesis. After that, an outlook will point at future directions which could be addressed based on the developments from this work.

6.1 Summary

The work of this thesis was dedicated to create a software framework with a high level approach to combine multiple software solvers from different scientific fields. The resulting software framework has been coined Multiscale Coupling Library and Environment (MUSCLE).

The foremost goal was to allow a maximum amount of flexibility: When bringing together computational models from different scientific fields, a coupling framework should provide solutions, not restrictions, to accomplish this task.

MUSCLE has been created as a so-called "agent based system", which per se imposes no restrictions on the individual software agents, as they are allowed to act autonomously. These agents are used to control the execution and remote communication of all the involved software solvers. To find a software library which could be utilised and extended to create the envisioned framework, this project has started with a literature survey and evaluation phase about existing libraries which could be used for the idea based on software agents.

A software framework has now been created, which allows to bring different kinds of software packages together and also configure the combined setup in a portable manner. In the resulting solution, the coupled sub-solvers are not restricted to a specific scientific field, numerical method, programming language or hardware resource. The connected solvers are allowed to span heterogeneous hardware and operating systems to be able to execute each sub-model in its preferred environment. MUSCLE can also be used to operate on large scale PC clusters to harness the capabilities of parallel computing. Software solvers with smaller code bases can easily be combined with full blown projects of the high performance community, which themselves require supercomputers to be executed.

The developed framework allowed to conduct several demanding coupled computations, one being a biomedical simulation, others involved fluid structure interactions. These have been performed on Linux PC clusters and also in settings where a single machine has been coupled to a vector computer.

MUSCLE has been the supporting framework of the successful EU project COAST and has been registered as an open source community project under terms of the GNU Lesser General Public License. It is being used by other European research groups, e. g. the MAPPER project, a recently started EU project [86] as well as the project Medical Devices Design in Cardiovascular Applications (MeDDiCA), which also aims to bring different scientific fields together [124]. MUSCLE has been set on the agenda of the VPH NoE toolkit (Virtual Physiological Human Network of Excellence) [170, 175]. The developed MUSCLE framework is also being used by the Grid Technologies Team of the AGH University of Science and Technology, Krakow.

The ideas of MUSCLE have been taken further to allow even more flexible coupling scenarios where the connections between the agents are determined automatically and can also change dynamically at runtime. This idea hit the limitations of the agent library JADE, which MUSCLE uses internally. For this reason the newly designed agent system (Bond) has been implemented, which allows this flexibility and has also been programmed with resource intensive high performance computing in mind. Bond allows fast and resource effective coupling scenarios on modern multi core hardware, and works also on large scale compute clusters.

Bond features a mechanism where new sub-solvers can be added (or removed) to a running simulation at any time. Bond automatically connects them to the existing coupling setup. This feature opens the path for many demanding coupling scenarios, where the number of coupling interactions can be too large to be determined manually.

6.2 Outlook

Software agents have their origins in artificial intelligence research and social behaviour simulations. The results from this work have shown that software agents provide interesting options for other scientific areas as well. Apart from their current usability, the developed frameworks MUSCLE and Bond can be extended further into two major directions:

Peta scale clusters

With the advent of more and more cluster systems which offer compute capabilities in the order of 10 P flop/s, the number of interconnected machines in these extreme scale clusters keeps growing. Additionally, the number of CPU cores per machine also increases rapidly. For example, the Blue Gene/Q supercomputer is planned for 2012 and should reach 20 P flop/s with 1.6 million cores on 98,304 boards [13].

Even cluster systems with "only" $\mathcal{O}(10^3)$ connected machines face the problem of frequent hardware failures, which will also occur while simulations are executed. Agent based systems like MUSCLE and Bond might be used to provide a kind of fault tolerant behaviour, which enables a distributed simulation to recover from a hardware failure and use spare machines to replace the defective ones. MUSCLE and Bond are already robust against any losses of machines, but the typical simulation will not be able to

continue automatically, as it can not recreate the lost sub-solvers. MUSCLE inherits an interesting feature from JADE: the agent monitoring can be executed redundantly on multiple machines, making it resilient against failures. Also, JADE agents can be migrated to other machines. Thus, if a hardware failure could be predicted, the agent could move with its sub-simulation to a safe machine.

In order to compete with the de facto standard message passing mechanism on cluster systems, MPI, a faster network communication could be implemented for Bond: Here, e. g. OpenFabrics could be used, which can utilise high speed networks like InfiniBand using the *sockets direct protocol* and avoids the TCP overhead [140, 154]. Another solution follows the techniques of Java Fast Sockets, as described in [158].

Self organising coupling scenarios

The concept of software models which may automatically establish connections to other models in a simulation, has already been realised with the developed Bond framework. However, especially for large automatic coupling scenarios there is room for improvement: The data still has to be mapped between sending and receiving agent. If there would be a common description for the data formats on both ends, the mapping procedure can probably be automatically determined to some degree. This could be achieved in a step by step approach using a filter system like the one introduced in MUSCLE. If the mapping can not be provided using a single existing filter, multiple filters can be combined automatically to convert the data to the inlet format. Standard meta data description models like the Resource Description Framework may provide means to express the required data format descriptions [143]. It shows some resemblance to the ontology definitions specified in the FIPA standard, which is already present in the JADE agent framework (section A.6).

Another area, from which all the above described coupling scenarios could benefit, would be to be able to create independent packages of coupling scenarios, which themselves can be coupled at a higher level. This could be done with e. g. an internal namespace mechanism, which clearly separated the communication belonging to a sub-package, but also allows agents from different packages to communicate directly (peer to peer).

The developed MUSCLE framework is already being used by other research groups, where it facilitates the development of multi science coupling scenarios in a natural manner. In terms of internal technology, the developed Bond framework is the successor of MUSCLE. Bond plays a major role in a recently submitted research proposal dealing with fault tolerant computations on extreme scale cluster computers.

A Appendix

A.1 Excerpt of EU Project COAST Description of Work

...
As the flexibility of the coupling platform is an important prerequisite for the success of the final demonstration prototype, it is mandatory to rely on modern software concepts supporting a multitude of communication patterns and hierarchical control structures. In recent years, so-called agent-based software approaches have been developed which operate on a synchronous as well as an asynchronous or event driven basis. Several platforms for distributed computations have emerged in the last decade (e.g. JADE, HLA, X-Machines) some of which are already FIPA standard compliant. Although matured to a certain degree, these systems are not yet sufficiently developed to fully support the Complex Automata coupling - and control - framework. Thus, after a literature survey and evaluation phase in the first part of the project, we will decide on the development environment. This platform will then be extended for the specific requirements, which are mandatory for the Complex Automata framework. This includes extensions to wrap the different CA-approaches by appropriate software layers and to provide flexible mechanisms to support distributed computations on heterogeneous hardware under special consideration of bandwidth and latency issues. The compatibility with other software libraries relevant for the project (e.g. MPI) and GRID-related features will be analysed in the early stage of this work package.
...

A.2 EU Project COAST Partners

- Universiteit van Amsterdam
- University of Sheffield
- Technische Universität Braunschweig
- Université de Genève
- NEC Europe Ltd.

A.3 JNI data types

Java type name	C type name	capacity [bits]
boolean	jboolean	8 (unsigned)
byte	jbyte	8 (signed)
char	jchar	16 (unsigned)
double	jdouble	64
float	jfloat	32
int	jint	32 (signed)
long	jlong	64 (signed)
short	jshort	16 (signed)

Table A.1: JNI primitive data types

typename	capacity [bits]	values/range
boolean	8	true or false
byte	8	-2^7 to 2^7-1 (-128 ... 127)
char	16	unicode symbol (0x0000 ... 0xFFFF)
double	64	$\pm 4.94065645841246544 \cdot 10^{-324}$ ... $\pm 1.79769131486231570 \cdot 10^{308}$
float	32	$\pm 1.40239846 \cdot 10^{-45}$ ... $\pm 3.40282347 \cdot 10^{38}$
int	32	-2^{31} to $2^{31}-1$ ($-2{,}147{,}483{,}648$ ... $2{,}147{,}483{,}647$)
long	64	-2^{63} to $2^{63}-1$ ($-9{,}223{,}372{,}036{,}854{,}775{,}808$ ... $9{,}223{,}372{,}036{,}854{,}775{,}807$)
short	16	-2^{15} to $2^{15}-1$ ($-32{,}768$ ... $32{,}767$)

Table A.2: Java primitive data types [168, 103]

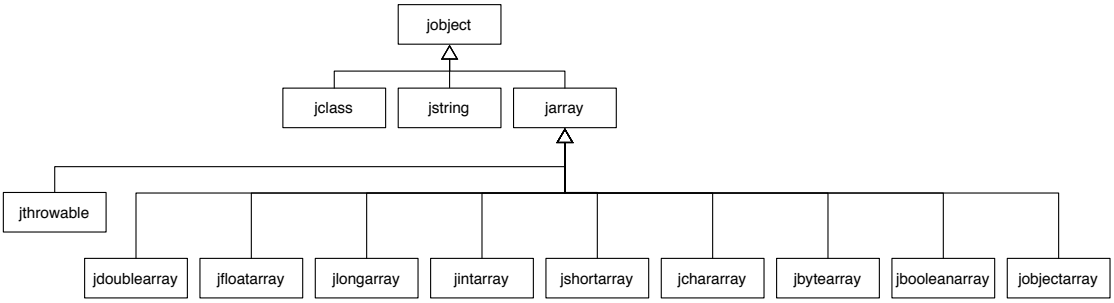


Figure A.1: Hierarchy of JNI C types

Java class	C type / C++ class
java.lang.Object	jobject
java.lang.Class	jclass
java.lang.String	jstring
n. a.	jarray
java.lang.Object[]	jobjectArray
boolean[]	jbooleanArray
byte[]	jbyteArray
char[]	jcharArray
short[]	jshortArray
int[]	jintArray
long[]	jlongArray
float[]	jfloatArray
double[]	jdoubleArray
java.lang.Throwable	jthrowable

Table A.3: Java references as C types

A.4 History of JADE

- JADE – Java Agent DEvelopment Framework
- by Telecom Italia (copyright holder)
- management supervision by JADE Board
  - Telecom Italia
  - Motorola
  - Whitestein Technologies AG.
  - Profactor GmbH
  - France Telecom R&D
- Open Source Software since February 2000
- current latest version of is JADE 4.0.1 released on 7th July 2010

JADE version history:

4.0.1	7 Jul 2010	3.3	1 Mar 2005	2.3	11 Jul 2001
4.0	20 Apr 2010	3.2	26 Jul 2004	2.2	11 Apr 2001
3.7	2 Jul 2009	3.1	17 Dec 2003	2.1	18 Dec 2000
3.6.1	4 Nov 2008	3.0b1	19 Mar 2003	2.01	19 Sep 2000
3.6	5 May 2008	2.61	24 Sep 2002	2.0	12 Sep 2000
3.5	25 May 2007	2.6	19 Jul 2002	1.4	6 Jun 2000
3.4.1	16 Nov 2006	2.5	5 Feb 2002	1.3	24 Feb 2000
3.4	14 Mar 2006	2.4	25 Sep 2001		

A.5 Other international projects using JADE

The following list is composed with information from [93]:

AgentCities initiative to create a next generation Internet that is based upon a worldwide network of services which work without user interaction ($\sim$100 companies and universities worldwide)

TeSCHeT enable the treatment and the organisation of the tourist-cultural information and the creation of spontaneous services networks (Telecom Italia Lab, Engineering Ingegneria Informatica S.p.A, Isufi)

PRIMO The PRIMO project deals with the design, the development, and the experimental validation of base stations and user terminals for wideband wireless communications systems able to cope with those reconfigurability and interoperability characteristics required by the next generation mobile communication systems.

IMAGE integrate existing georeference services (i.e. routing, mapping, proximity search, geo-coding, GPS tracking) and content providing services (lists of points of interest, e.g. restaurants, hotels, etc)

Pellucid multi-layer, agent-based system, based on JADE to aid employees in acquiring, reusing and sharing knowledge and experience

Knowledge on Demand design and build a multi-role personalised learning platform using collaborative software agents

CoMMa information management system supporting the consultation of a corporate memory

FACTS create a Personal Travel Assistant

MONADS extending existing systems with mobility-oriented features, in order to support the interaction of nomadic users wanting to access the fixed network services

DICEMAN create an agent-based market for digital audio and video trading

LIME dynamic user profiling, collective information dissemination and memory management

IM@GINE IT develop one single access point for existing geospatial info-mobility services, which are commonly only available locally

MicroGrids interconnection of small, modular generation sources to low voltage distribution systems to form a new type of power system

A.6 What is FIPA?

This is how the FIPA describes itself at [90]:

> *"FIPA was originally formed as a Swiss based organisation in 1996 to produce software standards specifications for heterogeneous and interacting agents and agent based systems. Since its foundations, FIPA has played a crucial role in the development of agents standards and has promoted a number of initiatives and events that contributed to the development and uptake of agent technology. Furthermore, many of the ideas originated and developed in FIPA are now coming into sharp focus in new generations of Web/Internet technology and related specifications."*

List of Figures

List of Listings

List of Tables

Bibliography

[1] *Agent Communication Language*. Foundation for Intelligent Physical Agents. 1997. URL: `http://www.fipa.org/specs/fipa00018/OC00018.pdf` (visited on 03/04/2011).

[2] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison Wesley, 2001, p. 352. ISBN: 0-201-70431-5.

[3] B. Allan, R. Armstrong, S. Lefantzi, J. Ray, E. Walsh, and P. Wolfe. *CCAFFEINE HOME*. 2005. URL: `http://www.cca-forum.org/ccafe/` (visited on 11/19/2010).

[4] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl. "The CCA core specification in a distributed memory SPMD framework". In: *Concurrency and Computation: Practice & Experience* 14 (5 2002), pp. 323–345.

[5] *Aman @ TU Braunschweig*. 2010. URL: `http://www.tu-braunschweig.de/irmb/institut/ausstattung/aman` (visited on 01/18/2011).

[6] G. M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *AFIPS 1967 Spring Joint Computer Conference, Atlantic City, New Jersey* (Apr. 1967), pp. 483–485.

[7] R. Armstrong, G. Kumfert, L. C. McInnes, S. Parker, B. Allan, M. Sottile, T. Epperly, and T. Dahlgren. "The CCA component model for high-performance scientific computing". In: *Concurrency and Computation: Practice & Experience - Computational Frameworks* 18 (2 Feb. 2006), pp. 215–229. ISSN: 1532-0626.

[8] L. Axner, J. Bernsdorf, T. Zeiser, P. Lammers, J. Linxweiler, and A. Hoekstra. "Performance evaluation of a parallel sparse lattice Boltzmann solver". In: *Journal of Computational Physics* 227.10 (2008), pp. 4895 –4911. ISSN: 0021-9991.

[9] H. Bauke and S. Mertens. *Cluster Computing*. Springer-Verlag Berlin Heidelberg, 2006. ISBN: 3-540-42299-4.

[10] N. H. F. Beebe. *ndiff. compare putatively similar files, ignoring small numeric differences*. 2000. URL: `http://www.math.utah.edu/~beebe/software/ndiff/` (visited on 10/06/2010).

[11] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. Ed. by M. Wooldridge. Wiley Series in Agent Technology. Wiley, 2007.

[12] J. Bloch. *Effective Java: A Programming Language Guide*. 2nd ed. Java Series. Addison-Wesley, 2008. ISBN: 978-0-321-35668-0.

[13] *Blue Gene - Wikipedia, the free encyclopedia*. URL: `http://en.wikipedia.org/wiki/Blue_Gene` (visited on 03/04/2011).

[14] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yechuri. "A component based services architecture for building distributed applications". In: *High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on*. 2000, pp. 51 –59.

[15] L. Braubach and A. Pokahr. *Overview (About.Overview) - XWiki*. URL: `http://jadex-agents.informatik.uni-hamburg.de/xwiki/bin/view/About/Overview` (visited on 03/04/2011).

[16] L. Braubach, A. Pokahr, and W. Lamersdorf. "Jadex: A BDI-Agent System Combining Middleware and Reasoning". In: *Software Agent-Based Applications, Platforms and Development Kits*. Ed. by M. Calisti, M. Walliser, S. Brantschen, M. Herbstritt, R. Unland, M. Calisti, and M. Klusch. Whitestein Series in Software Agent Technologies and Autonomic Computing. Birkhäuser Basel, 2005, pp. 143–168. ISBN: 978-3-7643-7348-1.

[17] B. Bügling. "The Component Template Library Protocol and its Java Implementation". Project Work. Institute of Scientific Computing, Technische Universität Braunschweig, Germany, 2006.

[18] S. Buis, A. Piacentini, and D. Déclat. "PALM: a computational framework for assembling high-performance computing applications". In: *Concurrency and Computation. Practice and Experience* 18 (2 Feb. 2006), pp. 231–245.

[19] F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*. Vol. 4. Software Design Patterns. Wiley, 2007, p. 636. ISBN: 978-0-470-05902-9.

[20] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1996. ISBN: 0-471-95869-7.

[21] *Cactus Code — Welcome*. URL: `http://www.cactuscode.org` (visited on 11/23/2010).

[22] A. Caiazzo, D. Evans, J.-L. Falcone, J. Hegewald, E. Lorenz, B. Stahl, D. Wang, J. Bernsdorf, B. Chopard, J. Gunn, R. Hose, M. Krafczyk, P. Lawford, R. Smallwood, D. Walker, and A. Hoekstra. "A Complex Automata approach for In-stent Restenosis: two-dimensional multiscale modeling and simulations". In: *Journal of Computational Science* (2010). in press.

[23] Canonical Ltd. *Server | Ubuntu*. 2010. URL: `http://www.ubuntu.com/server` (visited on 01/18/2011).

[24] R. H. Castain, T. S. Woodall, D. J. Daniel, J. M. Squyres, B. Barrett, and G. E. Fagg. "The Open Run-Time Environment (OpenRTE): A Transparent Multi-Cluster Environment for High-Performance Computing". In: *Proceedings, 12th European PVM/MPI Users' Group Meeting*. Sorrento, Italy, Sept. 2005.

[25] *CCA INtegration framework (CCAIN) | Download CCA INtegration framework (CCAIN) software for free at SourceForge.net*. URL: `http://sourceforge.net/projects/ccain/` (visited on 01/18/2011).

[26] *Ccaffeine Reference Framework for CCA Components*. URL: `http://www.cca-forum.org/ccafe/ccaffeine-man/` (visited on 01/18/2011).

[27] J. Cebral and R. Löhner. "On the loose coupling of implicit time-marching codes". In: *In 43rd AIAA Aerospace Sciences Meeting and Exhibit - Meeting Papers* (2005), pp. 10079–10093.

[28] *ChadFowler.com The Big Rewrite*. URL: `http://chadfowler.com/2006/12/27/the-big-rewrite` (visited on 08/31/2010).

[29] B. Chapman, G. Jost, and R. v. d. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. Ed. by W. Gropp, E. Lusk, and J. Kowalik. With a forew. by D. J. Kuck. The MIT Press, 2007. ISBN: 0262533022.

[30] S. Chapman and T. G. Cowling. *The Mathematical Theory of Non-uniform Gases*. Cambridge University Press, 1991.

[31] K. Chmiel, M. Gawinecki, P. Kaczmarek, M. Szymczak, and M. Paprzycki. "Efficiency of JADE agent platform". In: *Scientific Programming* 13.2 (Dec. 2005), pp. 159–172. ISSN: 1058-9244.

[32] B. Chopard, A. Masselot, and A. Dupuis. "A lattice gas model for erosion and particles transport in a fluid". In: *Computer Physics Communications* 129 (2000), pp. 167–176.

[33] P. Ciancarini and M. Wooldridge, eds. *Agent-Oriented Software Engineering, First International Workshop, AOSE 2000, Limerick, Ireland, June 10, 2000, Revised Papers*. Vol. 1957. Lecture Notes in Computer Science. Springer, 2001. ISBN: 3-540-41594-7.

[34] S. Coakley. "Formal Software Architecture for Agent-Based Modelling in Biology". PhD thesis. Department of Computer Science, University of Sheffield, 2007.

[35] S. Coakley, R. Smallwood, and M. Holcombe. "Using X-machines as a formal basis for describing agents in agent-based modelling". In: *Proceedings of 2006 Spring Simulation Multiconference*. Apr. 2006, pp. 33–40.

[36] Common Component Architecture Forum Tutorial Working Group. *A Hands-On Guide to the Common Component Architecture*. 0.7.1. 2010. URL: `http://www.cca-forum.org/download/tutorial/guide-0.7.1-0.pdf` (visited on 11/19/2010).

[37] *Component-based software engineering - Wikipedia, the free encyclopedia*. URL: http://en.wikipedia.org/wiki/Software_componentry (visited on 06/14/2010).

[38] *Contents*. URL: http://download.oracle.com/javase/6/docs/technotes/guides/jni/spec/jniTOC.html (visited on 12/09/2010).

[39] *Cougaar Agent Architecture*. URL: http://cougaar.org/ (visited on 11/30/2010).

[40] *CougaarReleases < Main < TWiki*. URL: http://cougaar.org/twiki/bin/view/Main/CougaarReleases (visited on 11/30/2010).

[41] J. S. Dahmann, R. M. Fujimoto, and R. M. Weatherly. "The Department of Defense High Level Architecture". In: *Proceedings of the 29th conference on Winter simulation*. WSC '97. IEEE Computer Society, 1997, pp. 142–149. ISBN: 0-7803-4278-X.

[42] *DARPA | Home*. URL: http://www.darpa.mil/ (visited on 11/30/2010).

[43] W. J. Davis and G. L. Moeller. "The High Level Architecture: Is there a better way?" In: *Proceedings of the 1999 Winter Simulation Conference*. Ed. by P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans. 1999.

[44] *Debian – The Universal Operating System*. Software in the Public Interest (SPI), Inc. Jan. 2011. URL: http://www.debian.org/ (visited on 01/18/2011).

[45] Defense Modeling and Simulation Office, United States Department of Defense. *http://www.msco.mil/StatusBoard.html*. (Visited on 01/28/2011).

[46] D. d'Humi'eres, I. Ginzburg, M. Krafczyk, P. Lallemand, and L.-S. Luo. "Multiple-relaxation-time lattice Boltzmann models in three-dimensions". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 360.1792 (2002), pp. 437–451.

[47] A. Dupuis. "From a lattice Boltzmann model to a parallel and reusable implementation of a virtual river". Dissertation. University of Geneva, 2002.

[48] B. Eckel. *Thinking in Java*. 4th ed. Upper Saddle River, NJ: Prentice Hall, 2006. ISBN: 978-0-13-187248-6.

[49] D. J. W. Evans, P. V. Lawford, J. Gunn, D. Walker, D. R. Hose, R. H. Smallwood, B. Chopard, M. Krafczyk, J. Bernsdorf, and A. Hoekstra. "The application of multiscale modelling to the process of development and prevention of stenosis in a stented coronary artery". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 366.1879 (Sept. 2008), pp. 3343–3360.

[50] C. A. Felippa, K. C. Park, and C. Farhat. "Partitioned analysis of coupled mechanical systems". In: *Computer Methods in Applied Mechanics and Engineering* 190.24-25 (2001), pp. 3247 –3270. ISSN: 0045-7825.

[51] R. T. Fielding. "Architectural styles and the design of network-based software architectures". Chair-Taylor, Richard N. PhD thesis. University of California, Irvine, 2000. ISBN: 0-599-87118-0.

[52] *FIPA-OS Agent Toolkit | Download FIPA-OS Agent Toolkit software for free at SourceForge.net.* URL: `http://sourceforge.net/projects/fipa-os/` (visited on 03/04/2011).

[53] *FLAME: What is FLAME?* URL: `http://www.flame.ac.uk` (visited on 06/11/2010).

[54] M. J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* c-21.9 (Sept. 1972), pp. 948–960.

[55] *Forschung::Projekte::CTL.* URL: `http://www.wire.tu-bs.de/forschung/projekte/ctl/e_ctl.html` (visited on 11/19/2010).

[56] P. S. Foundation. *Python Programming Language – Official Website.* URL: `http://www.python.org/` (visited on 02/19/2011).

[57] M. Fowler. *Domain-Specific Languages.* Addison-Wesley Professional, 2010. ISBN: 0321712943.

[58] S. Freudiger. "Entwicklung eines parallelen, adaptiven, komponentenbasierten Strömungskerns für hierarchische Gitter auf Basis des Lattice-Boltzmann-Verfahrens". Dissertation. Institut für Rechnergestützte Modellierung im Bauingenieurwesen, Technische Universität Braunschweig, 2009.

[59] R. Fujimoto. "Parallel and distributed simulation". In: *Winter Simulation Conference.* 1999, pp. 122–131.

[60] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[61] D. Gannon, R. Bramley, G. Fox, S. Smallen, A. Rossi, R. Ananthakrishnan, F. Bertrand, K. Chiu, M. Farrellee, M. Govindaraju, S. Krishnan, L. Ramakrishnan, Y. Simmhan, A. Slominski, Y. Ma, C. Olariu, and N. Rey-Cenvaz. "Programming the Grid: Distributed Software Components, P2P and Grid Web Services for Scientific Applications". In: *Cluster Computing* 5 (3 2002), pp. 325–336. ISSN: 1386-7857.

[62] S. Geller, M. Krafczyk, J. Tölke, S. Turek, and J. Hron. "Benchmark computations based on Lattice-Boltzmann, Finite Element and Finite volume Methods for laminar Flows". In: *Computers & Fluids* 35 (2006), pp. 888–897.

[63] S. Geller, M. Krafczyk, J. Tölke, S. Turek, and J. Hron. "Benchmark computations based on Lattice-Boltzmann, Finite Element and Finite Volume Methods for laminar Flows". In: *Elsevier Science* (2005).

[64] M. R. Genesereth and S. P. Ketchpel. "Software Agents". In: *Communications of the ACM* 37 (1994), pp. 48–53.

[65] D. Ghosh. *DSLs in Action*. 1st ed. Manning Publications, 2010, p. 376. ISBN: 9781935182450.

[66] R. Gustavsson. *Agent Oriented Software Engineering. A motivation for and an introduction to a novel approach to modeling and development of open distributed systems*. Research Report 5. Blekinge Institute of Technology, Karlskrona, Sweden, 1994.

[67] P. B. Hansen, ed. *The Origin of Concurrent Programming. From Semaphores to Remote Procedure Calls*. Springer-Verlag Berlin Heidelberg, 2002. ISBN: 0-387-95401-5.

[68] J. Hegewald. *Developers Guide to the MUSCLE*. URL: `http://muscle.berlios.de` (visited on 02/19/2011).

[69] J. Hegewald. *Developers Guide to the MUSCLE*. 2009. URL: `http://muscle.berlios.de/muscle_developers_guide.pdf` (visited on 01/11/2011).

[70] J. Hegewald. *MUSCLE download area*. URL: `http://developer.berlios.de/projects/muscle` (visited on 02/19/2011).

[71] J. Hegewald, M. Krafczyk, J. Tölke, A. G. Hoekstra, and B. Chopard. "An Agent-Based Coupling Platform for Complex Automata." In: *ICCS (2)*. Ed. by M. Bubak, G. D. van Albada, J. Dongarra, and P. M. A. Sloot. Vol. 5102. Lecture Notes in Computer Science. Springer, 2008, pp. 227–233. ISBN: 978-3-540-69386-4.

[72] A. Hejlsberg, M. Torgersen, S. Wiltamuth, and P. Golde. *The C# Programming Language*. 4th ed. Addison-Wesley Professional, 2010, p. 864. ISBN: 0321741765.

[73] A. Helsinger, M. Thome, and T. Wright. "Cougaar: a scalable, distributed multi-agent architecture". In: *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics 2004*. Vol. 2. 2004, pp. 1910 –1917.

[74] K. Henney, ed. *97 Things Every Programmer Should Know*. O'Reilly, 2010. ISBN: 978-0-596-80948-5.

[75] M. Henning. "A new approach to object-oriented middleware". In: *Internet Computing, IEEE* 8.1 (2004). Ed. by D. Lea and S. Vinoski, pp. 66 –75. ISSN: 1089-7801.

[76] M. Henning. *Choosing Middleware: Why Performance and Scalability do (and do not) Matter*. white paper. ZeroC, Inc., 2009.

[77] M. Henning and M. Spruiell. *Distributed Programming with Ice*. Version 3.4-IT1. ZeroC. Sept. 2010. URL: `http://www.zeroc.com/Ice-Manual.pdf` (visited on 11/19/2010).

[78] M. A. Heroux, P. Raghavan, and H. D. Simon. *Parallel Processing for Scientific Computing (Software, Environments and Tools)*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2006. ISBN: 0898716195.

[79] A. G. Hoekstra, B. Chopard, P. Lawford, R. Hose, M. Krafczyk, and J. Bernsdorf. "Introducing Complex Automata for Modelling Multi-Scale Complex Systems". In: *Proceedings of European Complex Systems Conference*. European Complex Systems Society, Oxford, UK, 2006.

[80] A. G. Hoekstra, E. Lorenz, J.-L. Falcone, and B. Chopard. "Toward a Complex Automata Formalism for MultiScale Modeling". In: *International Journal for Multiscale Computational Engineering* 5 (2007), pp. 491–502. ISSN: 1543-1649.

[81] A. G. Hoekstra, E. Lorenz, J.-L. Falcone, and B. Chopard. "Towards a Complex Automata Framework for Multi-scale Modeling: Formalism and the Scale Separation Map". In: *International Conference on Computational Science (1)*. Vol. 4487. Springer Verlag, 2007, pp. 922–930.

[82] R. Hoffmann, G. S. Mintz, G. R. Dussaillant, J. J. Popma, A. D. Pichard, L. F. Satler, K. M. Kent, J. Griffin, and M. B. Leon. "Patterns and Mechanisms of In-Stent Restenosis. A Serial Intravascular Ultrasound Study". In: *Circulation of the American Heart Association* 94.6 (Sept. 1996).

[83] M. Holcombe. "X-machines as a basis for dynamic system specification". In: *Software Engineering Journal* 3.2 (Mar. 1988), pp. 69 –76. ISSN: 0268-6961.

[84] M. Holcombe, S. Coakley, and R. Smallwood. "A General Framework for Agent-based Modelling of Complex Systems". In: *Proceedings of the European Conference on Complex Systems – ECCS. Towards a Science of Complex Systems* (2006).

[85] *Home – JRuby.org*. URL: `http://jruby.org/` (visited on 02/18/2011).

[86] *Home - mapper-project.eu*. 2010. URL: `http://www.mapper-project.eu/` (visited on 03/04/2011).

[87] *Home | Raytheon BBN Technologies*. URL: `http://www.bbn.com/` (visited on 11/30/2010).

[88] *HotSpot Group*. URL: `http://OpenJDK.Java.Net/groups/hotspot/` (visited on 06/14/2010).

[89] B. Hübner, E. Walhorn, and D. Dinkler. "A monolithic approach to fluid-structure interaction using space-time finite elements". In: *Computer Methods in Applied Mechanics and Engineering* 193.23-26 (2004), pp. 2087 –2104. ISSN: 0045-7825.

[90] IEEE Foundation for Intelligent Physical Agents, ed. *Welcome to the Foundation for Intelligent Physical Agents*. 2011. URL: `http://www.fipa.org/` (visited on 03/20/2011).

[91] G. Ingram, I. Cameron, and K. Hangos. "Classification and analysis of integrating frameworks in multiscale modelling". In: *Chemical Engineering Science* 59.11 (2004), pp. 2171–2187. ISSN: 0009-2509.

[92] *JADE - Java Agent DEvelopment Framework*. URL: `http://jade.tilab.com/` (visited on 03/20/2011).

[93] *JADE - Java Agent DEvelopment Framework*. URL: http://jade.tilab.com/application-projects.htm (visited on 03/20/2011).

[94] *jaRTI | Download jaRTI software for free at SourceForge.net*. URL: http://sourceforge.net/projects/jarti/ (visited on 03/04/2011).

[95] *Java SE Desktop Technologies - Java Beans*. URL: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138795.html (visited on 11/30/2010).

[96] N. R. Jennings. "An agent-based approach for building complex software systems". In: *Communications of the ACM* 44.4 (2001), pp. 35–41.

[97] *JScience*. URL: http://jscience.org/ (visited on 02/23/2011).

[98] *Kármán vortex street - Wikipedia, the free encyclopedia*. URL: http://en.wikipedia.org/wiki/Karman_vortex_street (visited on 03/20/2011).

[99] R. Karnapke, ed. *Lehrstuhl Verteilte Systeme/Betriebssysteme: Cocos*. URL: http://www.tu-cottbus.de/fakultaet1/de/betriebssysteme/forschung/projekte/cocos.html (visited on 01/18/2011).

[100] G. Karypis. *Family of Graph and Hypergraph Partitioning Software | Karypis Lab*. URL: http://glaros.dtc.umn.edu/gkhome/views/metis (visited on 03/20/2011).

[101] D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[102] C. Körner, T. Pohl, U. Rüde, N. Thürey, and T. Zeiser. "Parallel Lattice Boltzmann Methods for CFD Applications". In: *Numerical Solution of Partial Differential Equations on Parallel Computers, Lecture Notes in Computational Science and Engineering* 51 (2006), pp. 439–466. URL: http://www10.informatik.uni-erlangen.de/Publications/Papers/2005/LBMCFD_LNCSE51.pdf (visited on 09/2010).

[103] G. Krüger. *Handbuch der Java-Programmierung*. 3rd ed. Addison-Wesley, 2002.

[104] G. Krüger and T. Stark. *www.javabuch.de - Das Handbuch der Java-Programmierung*. URL: http://www.javabuch.de/ (visited on 10/01/2010).

[105] M. Krüger, R. Karnapke, and J. Nolte. "Controlling sensors and actuators collectively using the COCOS-framework". In: *Proceedings of the First ACM workshop on Sensor and actor networks*. SANET '07. 2007, pp. 53–54. ISBN: 978-1-59593-735-3.

[106] M. Krüger, R. Karnapke, and J. Nolte. "In-network Processing and Collective Operations using the COCOS-Framework". In: *12th IEEE Conference on Emerging Technologies and Factory Automation*. 2007.

[107] F. Kuhl, R. Weatherly, and J. Dahmann. *Creating Computer Simulation Systems. An Introduction to the High Level Architecture*. Prentice Hall PTR, 1999. ISBN: 0-13-022511-8.

[108] M. Lees, B. Logan, and G. Theodoropoulos. "Distributed Simulation of Agent-Based Systems with HLA". In: *ACM Transactions on Modeling and Computer Simulation* 17.3 (2007).

[109] *Lehrstuhl Verteilte Systeme/Betriebssysteme: Taco.* URL: `http://www.tu-cottbus.de/fakultaet1/en/distributed-systemsoperating-systems-group/research/projects/taco.html` (visited on 03/20/2011).

[110] A. D. Levin, N. Vukmirovic, C.-W. Hwang, and E. R. Edelman. "Specific binding to intracellular proteins determines arterial transport properties for rapamycin and paclitaxel". In: *Proceedings of the National Academy of Sciences of the United States of America* 101.25 (2004), pp. 9463–9467.

[111] S. Liang. *Java Native Interface: Programmer's Guide and Specification.* Prentice Hall, 1999. ISBN: 0201325772.

[112] J. Lindrud. *RCF - Interprocess Communication for C++ - CodeProject.* Dec. 2005. URL: `http://www.codeproject.com/KB/threads/Rcf_Ipc_For_Cpp.aspx` (visited on 09/17/2010).

[113] J. Lindrud. *RMI for C++ - CodeProject.* Feb. 2005. URL: `http://www.codeproject.com/KB/threads/RMI_For_Cpp.aspx` (visited on 09/17/2010).

[114] H. C. Lowe, S. N. Oesterle, and L. M. Khachigian. "Coronary in-stent restenosis: Current status and future strategies". In: *Journal of the American College of Cardiology* 39 (2002), pp. 183–193.

[115] G. C. Mai and C. A. F. De Rose. "Low cost cluster architectures for parallel and distributed processing". In: *CLEI Electronic Journal 3 (1)* (2000). URL: `http://www.clei.cl/cleiej/papers/v3i1p2.pdf` (visited on 02/19/2011).

[116] *Main Page - Jini.org.* URL: `http://www.jini.org/wiki/Main_Page` (visited on 03/20/2011).

[117] *Main Page - Portico.* URL: `http://porticoproject.org/index.php?title=Main_Page` (visited on 03/04/2011).

[118] D. Markovic, R. Niekamp, A. Ibrahimbegovic, H. G. Matthies, and R. L. Taylor. "Multi-scale modeling ofheterogeneous structures withinelastic constitutive behaviour. Part I – physical and mathematical aspects". In: *Engineering Computations: International Journal for Computer-Aided Engineering and Software* 22.5/6 (2005), pp. 664–683.

[119] P. Marrow, M. Koubarakis, R. van Lengen, F. Valverde-Albacete, E. Bonsma, J. Cid-Suerio, A. Figueiras-Vidal, A. Gallardo-Antolin, C. Hoile, T. Koutris, H. Molina-Bulla, A. Navia-Vazquez, P. Raftopoulou, N. Skarmeas, C. Tryfonopoulos, F. Wang, and C. Xiruhaki. "Agents in Decentralised Information Ecosystems: The DIET Approach". In: *Symposium on Information Agents for E-Commerce, AISB Convention.* University of York, United Kingdom, Mar. 2001.

[120] A. Masselot. "A New Numerical Approach to Snow Transport and Deposition by Wind: A Parallel Lattice Gas Model". Dissertation. University of Geneva, 2000.

[121] H. G. Matthies, R. Niekamp, and J. Steindorf. "Algorithms for strong coupling procedures". In: *Computer Methods in Applied Mechanics and Engineering* 195.17-18 (2006), pp. 2028 –2049. ISSN: 0045-7825.

[122] R. McHaney. *Computer simulation: a practical perspective*. San Diego, CA, USA: Academic Press, Inc., 1991. ISBN: 0-12-484140-6.

[123] G. R. McNamara and G. Zanetti. "Use of the Boltzmann Equation to Simulate Lattice-Gas Automata". In: *Physical Review Letters* 61.20 (Nov. 1988), pp. 2332–2335.

[124] *MeDDiCA.EU*. URL: `http://www.meddica.eu/` (visited on 03/04/2011).

[125] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*. 1997. URL: `http://www.mpi-forum.org/docs/mpi-20.ps` (visited on 09/17/2010).

[126] *MetaMPICH - Flexible Coupling of Heterogenous MPI Systems*. URL: `http://www.lfbs.rwth-aachen.de/~martin/MetaMPICH/` (visited on 11/18/2010).

[127] H. W. Meuer. "The TOP500 Project: Looking Back Over 15 Years of Supercomputing Experience". In: *Informatik-Spektrum* 31 (3 2008), pp. 203–222. ISSN: 0170-6012.

[128] *Middlesim | Download Middlesim software for free at SourceForge.net*. URL: `http://sourceforge.net/projects/middlesim/` (visited on 03/04/2011).

[129] *MPICH2 : High-performance and Widely Portable MPI*. URL: `http://www.mcs.anl.gov/research/projects/mpich2/` (visited on 03/04/2011).

[130] G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas. *The Art of Software Testing*. 2nd ed. John Wiley & Sons, Inc., 2004, p. 256. ISBN: 978-0-471-46912-4.

[131] *Myrinet Overview*. Myricom, Inc. 2009. URL: `http://www.myri.com/myrinet/overview/` (visited on 01/18/2011).

[132] B. Nachtwey. *Paralleles und Verteiltes Rechnen*. Vorlesungsumdruck. Institut für Computeranwendungen im Bauingenieurwesen, TU-Braunschweig, 2004.

[133] J. Newmarch. *Foundations of Jini 2 Programming*. Apress, 2006. ISBN: 1590597168.

[134] J. Nolte, M. Sato, and Y. Ishikawa. "TACO - Template Based Collections for Distributed Computing Platforms". In: *The Fourth International Conference on High-Performance Computing in the Asia-Pacific Region – HPC Asia* (May 2000), pp. 463–469.

[135] C. O. Nutter, T. Enebo, N. Sieger, O. Bini, and I. Dees. *Using JRuby*. Bringing Ruby to Java. Ed. by J. Carter. 2011, p. 300. ISBN: 978-1-934356-65-4.

[136] *Object Management Group*. URL: `http://www.omg.org/` (visited on 11/19/2010).

[137] *Obtaining ACE, TAO, CIAO, and DAnCE*. URL: http://download.dre.vanderbilt.edu/ (visited on 11/19/2010).

[138] R. Oechsle. *Parallele und verteilte Anwendungen in Java*. Hanser Fachbuchverlag, 2007. ISBN: 3446407146.

[139] *OMG IDL / Language Mappings Specifications*. 2010. URL: http://www.omg.org/technology/documents/idl2x_spec_catalog.htm (visited on 11/19/2010).

[140] OpenFabrics Alliance. *The OpenFabrics Alliance - Open-source software stack for Linux and Windows*. 2010. URL: http://www.openfabrics.org/ (visited on 03/04/2011).

[141] B. Page. *Diskrete Simulation. Eine Einführung mit Modula-2*. Springer Verlag Berlin, 1991.

[142] *Pitch - Training*. URL: http://www.pitch.se/training (visited on 01/31/2009).

[143] *Resource Description Framework - Wikipedia, the free encyclopedia*. URL: http://en.wikipedia.org/wiki/Resource_Description_Framework (visited on 03/04/2011).

[144] A. Rubin. *Multi-Simulation Interface (MSI) Brochure*. 2006. URL: http://msi.sourceforge.net/ (visited on 03/20/2011).

[145] *Ruby Programming Language*. URL: http://www.ruby-lang.org/en/ (visited on 02/18/2011).

[146] K. J. Rycerz. "Grid-based HLA Simulation Support". PhD thesis. University of Amsterdam and AGH Krakow, 2006.

[147] D. C. Schmidt. *Real-time CORBA with TAO (The ACE ORB)*. 2010. URL: http://www.cs.wustl.edu/~schmidt/TAO.html (visited on 11/19/2010).

[148] D. C. Schmidt, H. Rohnert, M. Stal, and D. Schultz. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. New York, NY, USA: John Wiley & Sons, Inc., 2000. ISBN: 0471606952.

[149] R. S. Schwartz, A. Chu, W. D. Edwards, S. S. Srivatsa, R. D. Simari, J. M. Isner, and D. R. Holmes. "A proliferation analysis of arterial neointimal hyperplasia: lessons for antiproliferative restenosis therapies". In: *International Journal of Cardiology* 53 (1 1996), pp. 71 –80. ISSN: 0167-5273.

[150] *SCIRun*. URL: http://www.sci.utah.edu/cibc/software/106-scirun.html (visited on 01/11/2011).

[151] J. Shore. "Fail Fast". In: *IEEE Software* 21 (2004), pp. 21–25. ISSN: 0740-7459.

[152] Silicon Graphics International. *SGI - Products: Servers and Supercomputers: SGI Altix 4700: Features and Benefits*. URL: http://www.sgi.com/products/servers/altix/4000/features.html (visited on 03/04/2011).

[153] Simulation Interoperability Standards Committee (SISC) of the IEEE Computer Society. *Standard for Modeling and Simulation High Level Architecture. Federate Interface Specification*. IEEE Standard 1516.1-2000. 2000.

[154] *Sockets Direct Protocol - Wikipedia, the free encyclopedia*. URL: `http://en.wikipedia.org/wiki/Sockets_Direct_Protocol` (visited on 03/04/2011).

[155] J. Spolsky. *Joel on Software. And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity*. Apress, 2004, p. 362. ISBN: 1590593898.

[156] *Standard ECMA-334*. URL: `http://www.ecma-international.org/publications/standards/Ecma-334.htm` (visited on 02/18/2011).

[157] M. Stiebler, J. Tölke, and M. Krafczyk. "Advection-diffusion lattice Boltzmann scheme for hierarchical grids". In: *Computers and Mathematics with Applications* 55.7 (2008), pp. 1576–1584. ISSN: 0898-1221.

[158] G. L. Taboada, J. Touriño, and R. Doallo. "Java Fast Sockets: Enabling high-speed Java communications on high performance clusters". In: *Computer Communications* 31 (17 Nov. 2008), pp. 4049–4059. ISSN: 0140-3664.

[159] B. Technologies. *Cougaar Architecture Document*. 2004. URL: `http://cougaar.org/twiki/pub/Main/CougaarDocuments/CAD_11_4.pdf` (visited on 11/30/2010).

[160] *The Common Component Architecture Forum*. URL: `http://www.cca-forum.org/` (visited on 11/19/2010).

[161] *The Maxine Virtual Machine*. URL: `http://labs.oracle.com/projects/maxine/` (visited on 06/11/2010).

[162] *The O-PALM dynamic parallel coupler*. URL: `http://www.cerfacs.fr/globc/PALM_WEB/` (visited on 03/20/2011).

[163] The Open MPI Project. *Open MPI: Open Source High Performance Computing*. URL: `http://www.open-mpi.org/` (visited on 03/20/2011).

[164] *The XCAT Project*. 2004. URL: `http://www.extreme.indiana.edu/xcat/` (visited on 03/20/2011).

[165] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby 1.9: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2009. ISBN: 1934356085.

[166] A. Travers. *SourceForge.net: ASH (A Simple HLA) middleware - Project Web Hosting - Open Source Software*. URL: `http://ash.sourceforge.net/` (visited on 11/19/2010).

[167] C. Ullenboom. *Galileo Computing :: Java ist auch eine Insel (8. Auflage)*. URL: `http://openbook.galileocomputing.de/javainsel8/` (visited on 11/30/2010).

[168] C. Ullenboom. *Java ist auch eine Insel*. Ed. by G. Computing. 8th ed. Galileo Computing, 2009. Chap. 28, p. 1475.

[169] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. 2002, p. 552. ISBN: 0-201-73484-2.

[170] *VPH NoE ToolKit*. 2009. URL: `http://toolkit.vph-noe.eu/` (visited on 03/04/2011).

[171] W. A. Wall, S. Genkinger, and E. Ramm. "A strong coupling partitioned approach for fluid-structure interaction with free surfaces". In: *Computers & Fluids* 36.1 (2007). Challenges and Advances in Flow Simulation and Modeling, pp. 169 –183. ISSN: 0045-7930.

[172] E. Weinan, B. Engquist, X. Li, W. Ren, and E. Vanden-Eijnden. "Heterogeneous Multiscale Methods: A Review". In: *Communications in Computational Physics* 2.3 (June 2007), pp. 367–450.

[173] G. Weiss, ed. *Multiagent systems. A Modern Approach to Distributed Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1999. ISBN: 0-262-23203-0.

[174] M. Wooldridge. *An Introduction to MultiAgent Systems*. 2nd ed. John Wiley & Sons, Inc., 2009. ISBN: 0470519460.

[175] *WP3 - VPH ToolKit*. 2010. URL: `http://www.vph-noe.eu/wp3` (visited on 03/04/2011).

[176] *XStream - About XStream*. URL: `http://xstream.codehaus.org/` (visited on 12/06/2010).

[177] ZeroC, Inc. *Welcome to ZeroC, the Home of Ice*. 2010. URL: `http://www.zeroc.com/index.html` (visited on 11/19/2010).

[178] K. Zhang, K. Damevski, and S. G. Parker. "SCIRun2: A CCA framework for high performance computing". In: *Proceedings of the Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*. IEEE Press, 2004, pp. 72–79.